NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

REAL-TIME SONAR CLASSIFICATION FOR AUTONOMOUS UNDERWATER VEHICLES

by

Michael Scott Campbell

March 1996

Thesis Advisors: Don Brutzman
Xiaoping Yun

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1996	Master's Thesis

4. TITLE AND SUBTITLE
Real-Time Sonar Classification for Autonomous Underwater Vehicles (U)

5. FUNDING NUMBERS

6. AUTHOR(S)
Campbell, Michael S.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, California 93943-5000 USA

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Naval Postgraduate School autonomous underwater vehicle (AUV) Phoenix did not have any sonar classification capabilities and only a basic collision avoidance system. The Phoenix also did not have the capability of dynamically representing its environment for path planning purposes.

This thesis creates a sonar module that handles real-time object classification and enables collision avoidance at the Tactical level. The sonar module developed communicates directly with the available sonar and preprocesses raw data to a range/bearing data pair. The module then processes the range/bearing data using parametric regression to form line segments. A polyhedron-building algorithm combines line segments to form objects and classifies them based on their attributes. When the Phoenix is transiting, the classifying algorithm detects collision threats and initiates collision avoidance procedures.

The result of this thesis is a fully implemented sonar module on the Phoenix. This module was tested in a virtual world, test tank and in the first ever sea-water testing of the Phoenix. The sonar module has demonstrated real-time sonar classification, run-time collision avoidance and the ability to dynamically update the representation of the unknown environment. The sonar module is a forked process written in the "C" language, functioning at the Tactical level. Source code and output from an actual Phoenix mission displaying the object classification of the sonar module are included.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Autonomous underwater vehicle, obstacle avoidance, sonar sensing, real-time sonar classification, mine countermeasures, parametric regression	118
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

REAL-TIME SONAR CLASSIFICATION FOR AUTONOMOUS UNDERWATER VEHICLES

Michael Scott Campbell
Lieutenant, United States Navy
B.S.E.E., Ohio Northern University, 1989

Submitted in partial fulfillment of the
requirements for the degrees of

MASTER OF SCIENCE IN COMPUTER SCIENCE

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL

March 1996

ABSTRACT

The Naval Postgraduate School autonomous underwater vehicle (AUV) Phoenix did not have any sonar classification capabilities and only a basic collision avoidance system. The Phoenix also did not have the capability of dynamically representing its environment for path planning purposes.

This thesis creates a sonar module that handles real-time object classification and enables collision avoidance at the Tactical level. The sonar module developed communicates directly with the available sonar and preprocesses raw data to a range/ bearing data pair. The module then processes the range/bearing data using parametric regression to form line segments. A polyhedron-building algorithm combines line segments to form objects and classifies them based on their attributes. When the Phoenix is transiting, the classifying algorithm detects collision threats and initiates collision avoidance procedures.

The result of this thesis is a fully implemented sonar module on the Phoenix. This module was tested in a virtual world, test tank and in the first ever sea-water testing of the Phoenix. The sonar module has demonstrated real-time sonar classification, run-time collision avoidance and the ability to dynamically update the representation of the unknown environment. The sonar module is a forked process written in the "C" language, functioning at the Tactical level. Source code and output from an actual Phoenix mission displaying the object classification of the sonar module are included.

TABLE OF CONTENTS

I. INTRODUCTION

A. MOTIVATION

Mine warfare has long been a major challenge for the United States Armed Forces. Current mine countermeasure (MCM) methods, which are sometimes as rudimentary as a sailor with an M-14 rifle, are unacceptable and need to be improved. In a recent White Paper "Mine Countermeasures - An Integral Part of Our Strategy and Our Forces," Chief of Naval Operations J. M. Boorda cited the recent damages to the USS Samuel B. Roberts (FFG-58), Tripoli (LPH-10), and Princeton (CG-59) to demonstrate the threat of naval mines (Boorda 96). The cost of the damages ($125 million) compared to the cost of the mines (approximately $30 thousand) demonstrates the need for better research and development of results in MCM. Because of the desirability of using robots to perform undesirable and dangerous tasks, autonomous underwater vehicles (AUVs) are an attractive possibility to the mine hunting problem in shallow waters.

Many capabilities are required for an AUV to support mine hunting. At a minimum an AUV must be able to perform real-time sonar classification and demonstrate run-time collision avoidance. The abilities to detect, localize and classify unknown objects are essential requirements for this mission. Collision detection and collision avoidance are required for the safe operation of an AUV in unknown waters.

The Naval Postgraduate School AUV (named Phoenix) is an ideal platform for shallow water mine hunting experiments. The current internal design of the Phoenix is shown in Fig. 1.1, and an external drawing is displayed in Fig. 1.2. The Phoenix has demonstrated the ability to operate untethered in an unknown environment, enabling researchers to conduct complex experiments in mine hunting.

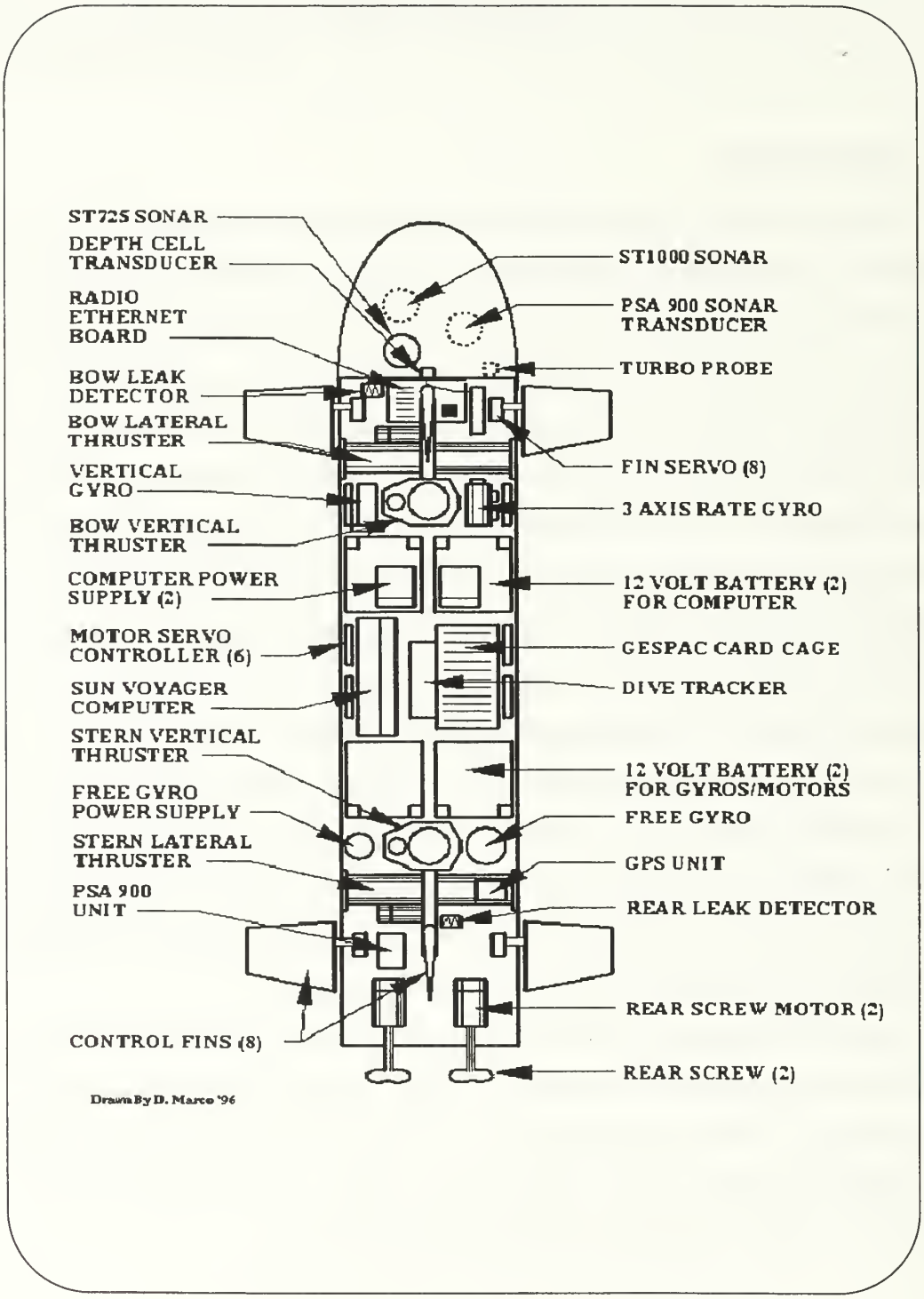

Figure 1.1. Internal View of Phoenix (Marco 96).

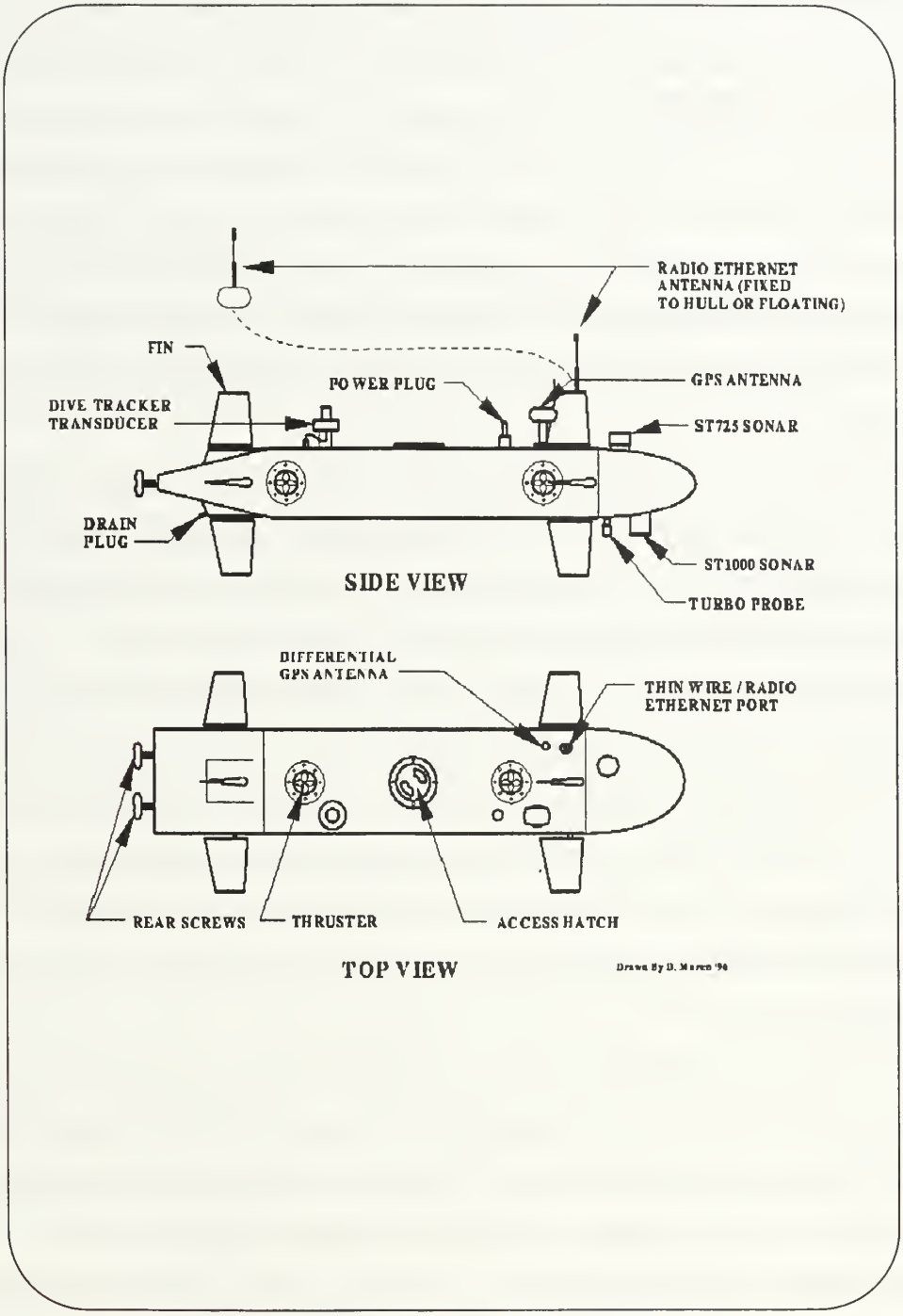

Figure 1.2. External Views of Phoenix (Marco 96).

B. PHOENIX PROJECT

1. Software Architecture

The Phoenix is controlled by a tri-level architecture called the *Rational Behavior Model* (RBM) (Byrnes 93). RBM consists of a top level that is entirely symbolic with no global variables, a bottom level that is synchronous and numerically intensive, and a middle level which provides analytic modules and interfaces between the other two levels. The current implementation also contains some of the hardware control at the middle level.

The highest level of the model is the Strategic level, this level has the mission plan and controls the actions of the vehicle. The Strategic level ensures that the mission is completed to the greatest extent possible, by passing commands to the Tactical level.

The Tactical level is made up of numerous processes written in the "C" language, the main process is the Officer of the Deck (OOD) module. The OOD is the only module that communicates with the Strategic and Execution levels. The OOD forks all of the other Tactical level modules, the Navigator module, the Replanner module, and the Sonar module. The OOD then passes commands to these processes as required by the current phase of the mission. The OOD module also processes the data from the forked processes and initiates the required actions, i.e., orders to the Execution level or responses to the Strategic level.

The Execution level directly controls the hardware based on the orders from the Tactical level. Real-time constraints required for the stability of the AUV are all handled at this level. Many safety features are also built into this level that might cause a fail-safe abort to the surface. The automatic abort situations include flooding, loss of communications with the Tactical level, loss of DiveTracker acoustic navigation and loss of depth control.

Pipes are used for interprocess communication (IPC) on the Tactical level. Communication between the Strategic level and Tactical level is done with function calls

and returned values. Communication between the Execution level and Tactical level is accomplished using sockets. The Sonar module and the Navigator module communicate with their respective sensor hardware through serial ports. A communications diagram appears in Fig 2.1.

2. Hardware Systems

The current hardware in the vehicle consists of a Gespac M68030 series computer system and a Sun Voyager Sparc 5 workstation. The OS-9 operating system is used on the Gespac providing the real-time multitasking needed by the Execution level for hydrodynamic control stability. The Tactical and Strategic levels run on the Sun Voyager under SunOS 5.4. The two computer systems form a LAN through Ethernet connections within the vehicle, which can also be (optionally) networked through an external Ethernet connection. Communications between systems is done through software sockets. The Sonar and Navigator modules communicate with peripherals through serial ports, one of which is used directly by the Voyager. Other serial ports connect through a SCSI serial interface which increases the number of remaining serial ports available for future use.

The available sonar systems are a Tritech ST725, which is a 750 kHz scanning sonar and a Tritech ST1000, a 1250 kHz profiling sonar [Tritech 92]. The ST725 has a one degree wide by 24 degree vertical fan beam. This beam is steerable with azimuth rotation step sizes of 0.9, 1.8 and 2.6 degrees. The range options for the ST725 are one, two, four, six, ten, 20, 25, 30, 50 and 100 meters. The ST1000 sonar can operate both in scanning mode and profiling mode. The ST1000 transmits a one degree conical beam. The range scale consists of eight possible selections ranging from three to 160 feet. Both sonar systems are steerable, thus providing 360 degree coverage. Some "baffling" (i.e. occlusion) of sonar signals is possible when pointing astern due to returns from the vehicle.

C. THESIS OBJECTIVES

The objective of this work was to create a Sonar module that operates at the Tactical level where it receives orders from the OOD module and conducts sonar searches based on those orders. This module communicates directly with the sonar systems to collect raw data, processes the raw data to perform real-time sonar classification and produces the messages required to achieve run-time collision avoidance. To achieve these goals, the information developed by the Sonar module in the form of "new worlds" (i.e. circle models) and "collision threats" is passed to the OOD to initiate the required actions. A description of Phoenix sonar operations follows.

The modes of operation now available for Sonar are "Transit Search," "Sonar Search" and "Rotate Search." Transit Search is performed when the Phoenix is transiting between waypoints, continuously scanning the sonar between 325 and 035 degrees relative bearing. The main purpose of Transit Search is collision avoidance. The Sonar Search and Rotate Search are used when the Phoenix is stationary and a search of the surrounding area is desired. These searches are used to locate and classify unknown objects. The Sonar Search is a 360 degree rotation of the sonar with the vehicle heading fixed, while the Rotate Search is done with the sonar head fixed and a 360 degree rotation of the vehicle.

Sonar classification begins with the preprocessing of the raw sonar data The resulting processed returns are then fitted to line segments using parametric regression. Line segments are then combined based on proximity and orientation to form polyhedra. The polyhedra are classified based on their characteristics. The classified objects are then represented as circles in a world file, which is then used by the Replanner to plan the paths between waypoints.

Collision avoidance is accomplished by evaluating the range and bearing of each sonar return during the building of line segments. When collision threats are discovered they are passed to the OOD, permitting collision avoidance actions to be taken. The

current execution level collision avoidance actions, which merely backs down until headway is removed, can easily be improved to take less drastic measures.

D. SUMMARY

The recent losses the United States Navy has experienced due to mine warfare has shown the urgent need to improve MCM. AUVs are one of the potential platforms that could be used for mine-hunting in very shallow water. The Naval Postgraduate School Phoenix AUV has progressed to the point where it is an ideal platform for AUV experiments in shallow-water mine-hunting.

The tri-level architecture of the Phoenix can be compared to the human chain of command on board a real submarine with the Strategic level corresponding to the Commanding Officer (CO), the Tactical level corresponding to the supervisory watches, and the Execution level corresponding to the watchstanders in the spaces who operate the equipment. The Sonar module create by this work represents the sonar watchstanders who analyze the raw data and the sonar supervisor who evaluates their results and reports to the OOD.

II. PREVIOUS WORK

A. INTRODUCTION

Previous work on the Phoenix project has created a software architecture paradigm called the Rational Behavior Model (RBM) (Byrnes 93). Other previous work related to the Phoenix project includes an expert system for sonar classification (Brutzman 92) and a circle world replanner (Brutzman 92). All of these works have been improved and reimplemented to advance the Phoenix project. The virtual world created by (Brutzman 94) also played a vital role in the development of the software modules for the Phoenix.

B. TRI-LEVEL ARCHITECTURE

1. Paradigm

The current implementation of the RBM is represented in Fig. 2.1. This control architecture resembles the command structure of a manned submarine. The strategic level corresponds to the Commanding Officer (CO), the tactical level corresponds to the OOD and watch officers, and the execution level corresponds to the actual equipment and watchstanders operating the equipment.

2. Strategic Level

This level controls the overall condition of the vehicle through planning and deciding on operational tasks, and then giving orders to the OOD. These orders are based on the current phase of the mission. Responses from the Tactical level are evaluated to determine the status for the current phase, as well as determining whether to proceed to another phase or to abort the mission. The Strategic level is currently implemented in Prolog.

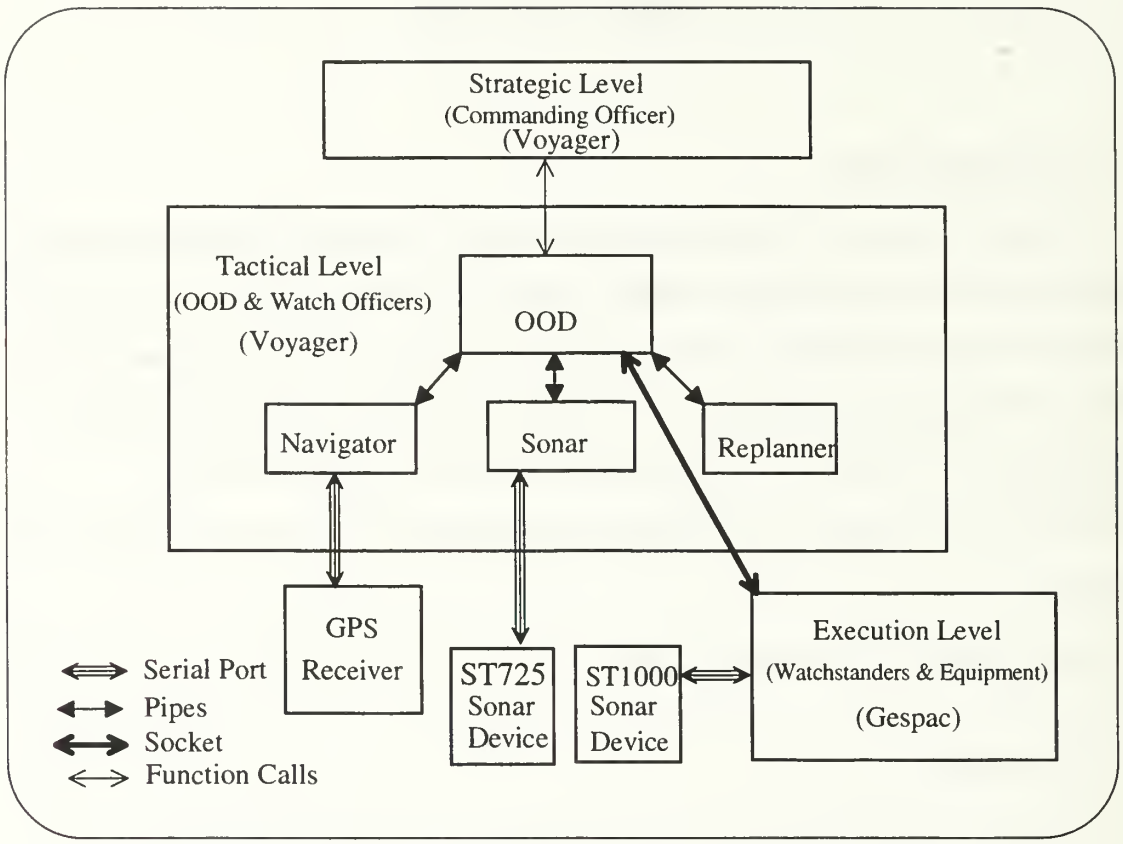

Figure 2.1 Current Software Architecture and Communication Modes.

3. Tactical Level

This level is run by the OOD module, which forks the other modules at startup. The OOD is the only module that communicates with the other levels, via sockets with the Execution level and via function calls and returned values with the Strategic level. All communications within the Tactical level are done via pipes. Since there is no shared memory established in the current implementation, all data that is required by more than one module is either passed in message format or written to files to be accessed at a later time by other modules. The modules that are currently operating at this level are the OOD module, the Sonar module, the Navigator module, and the Replanner module. All processes in this level are implemented in the "C" language.

4. Execution Level

The Execution level alone resides on the Gespac system. This level is currently implemented in the "C" language and is written to work both onboard the Phoenix and in the virtual world. The Execution level handles the control and stability of the Phoenix through the control of all of the control planes, thrusters, and screws. This level also implements many safety features to perform an abort script that aborts the mission by surfacing the AUV. This abort script can be triggered by low battery voltage, flooding, loss of depth control, or loss of tactical level communications.

C. EXPERT SYSTEM FOR SONAR CLASSIFICATION

Previous work developed an expert system for sonar classification (Brutzman 92). The sonar classification expert system was written in the Clips language (NASA 91) and processed sonar data offline due to the computational demands of the expert system. The starting point of this thesis was to convert that expert system into an onboard real-time system. The existing expert system uses parametric regression to line fit the sonar data. A sliding window is used to locate a suitable starting point for a line segment. Line segments are combined based on time sequence, distance and orientation. The combined line segments build polyhedra which are then classified based on their characteristics.

D. CIRCLE WORLD REPLANNER

Current work has developed a Replanner module at the Tactical level (Leonhardt 96) . This module creates a safe path between the AUV's current location and the desired location, using the "circle_world.inputX" file created by the Sonar module. The input file has the format of "Circle x-position y-position z-position radius." This module was derived from the work of (Brutzman 92). The Replanner uses the circle world representation of the obstacles, the start point, and the goal point to create a file containing segments and arcs which can be used to safely traverse the obstacle field.

E. VIRTUAL WORLD

The virtual world created by (Brutzman 94) is an invaluable asset in the development of new software for the Phoenix. The virtual world allows for initial testing of new software and software modifications without the vehicle being deployed in water.

F. CURRENT WORK

The Phoenix project has made significant advancements during the past six months. Many of these improvements are discussed in this thesis and (Leonhardt 96). Other recent work includes the combination of the virtual world's and Phoenix's Execution levels to form a single execution program that works in both environments (Burns 96). The installation of DiveTracker acoustic navigation on the Phoenix was part of the work by (Scrivener 96), the integration of the DiveTracker and the GPS data into the Navigator module was accomplished by (McClarin 96).

G. SUMMARY

The use of autonomous vehicles for jobs that are either undesirable or dangerous for human beings is the driving force behind many robotics research projects. MCM is an area that is ideal for robots. Many organizations have been working on the issue of autonomy for robots for many years. With the work of this thesis and the many works cited above, the Phoenix has demonstrated the ability to operate in, and interact with, an unknown environment. These capabilities allow for further testing and software development in the support of mine-hunting solutions. It is now clear that AUVs are on the threshold of effectively performing minefield search missions.

III. PROBLEM STATEMENT

A. INTRODUCTION

Real time sonar classification and run time collision avoidance are mandatory requirements for truly autonomous operation in unknown environments. To achieve these objectives, the process starts with the gathering and processing of raw sonar data, including the initialization the sonar system. The next problem is to create object representations from the processed sonar data using polyhedra. Once the polyhedra are built, object classification occurs. The final step in the classification process is the representation of the objects in a format that allows for path planning. The flow of data is shown in Fig. 3.1.

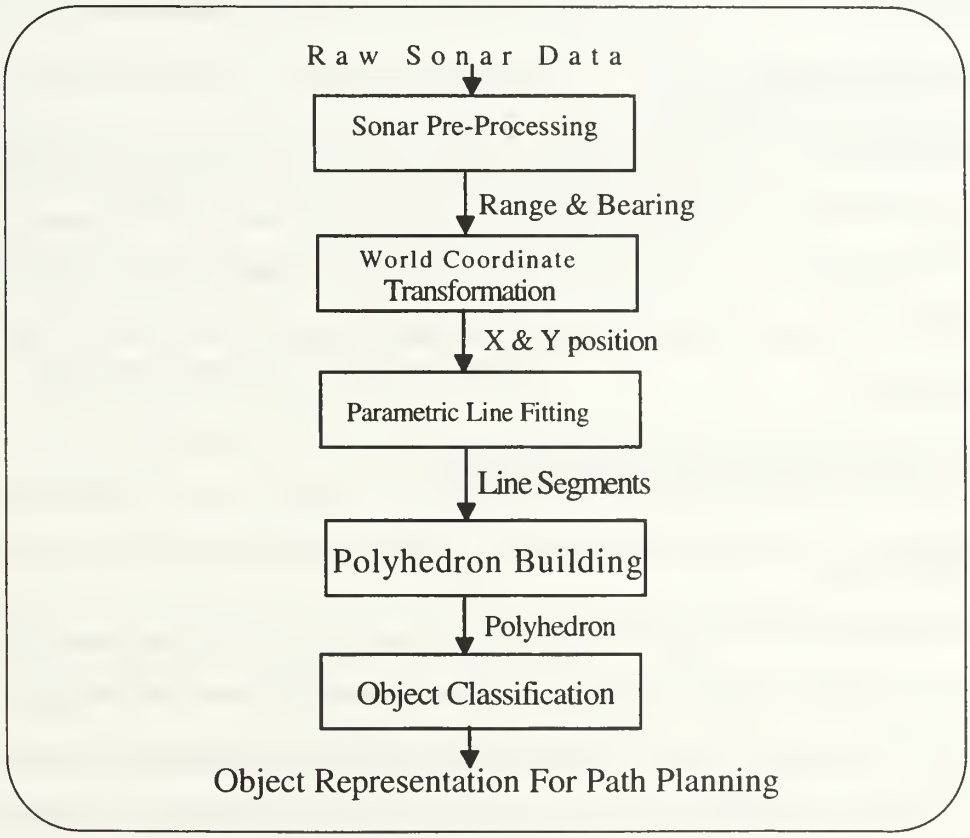

Figure 3.1 Sonar Classification Data Flow Chart.

The collision avoidance task is essentially independent of sonar classification. Collision avoidance can be broken into two problems: first identifying the existence of collision threat, and second reacting both properly and promptly to avoid a collision.

B. SONAR DATA PROCESSING

1. ST725 Scanning Sonar

Sonar processing begins with the initialization of the sonar head. This step is where the maximum range, azimuth rotation step size, receiver gain, and transmitter power parameters are set. The second step in the process is to analyze the data returned by the sonar head. The ST725 sonar returns a 33 byte string, with 32 bytes of the string representing the strength (between zero and fifteen) of the sonar return over the range scale divided into 64 bins. The preprocessing of this ping return data must produce a single range/bearing pair, to be further analyzed by the classification algorithm.

2. ST1000 Profiling Sonar

The ST1000 sonar can be operated as a scanning sonar, like the ST725, with 64 or 128 bins thus presenting the same preprocessing requirements that the ST725 sonar presents. The ST1000 can also be operated in profiling mode, where the return is a range in mm. ST1000 profiling mode does not require range/bearing postprocessing as the scanning mode does.

C. REAL-TIME SONAR CLASSIFICATION

1. Line Fitting

The first step of the classification problem begins with fitting the processed sonar data into line segments. The line fitting problem consists of starting the line segment, adding to the line segment once started, and when to finish the line segment. The ability to handle spurious and intermittent returns is a problem that also needs to be addressed while fitting lines to the data.

2. Polyhedron Building

Once line segments are formed the problem becomes how and when to combine the line segments to build objects. Our approach is to first create cylindrical polyhedra. The storage of objects once they have been formed, presents problems regarding what data structures are needed and what information needs to be maintained?

3. Classification of Objects

The final step in the process is the actual classification of the object. This step involves determination of what characteristics should be used for classification, and how the characteristics will be used for classification. Determining when an object should be classified is also an important issue: should the object be classified during the building process or only when an object is completed.

4. Representation of Obstacles

The final step in the classification process is how to represent the objects for path planning purposes. This includes the problem of how to format this information and how to share it with the OOD to support autonomous path planning.

D. OBSTACLE AVOIDANCE

1. When to Check for Collision Threats

The first question in obstacle avoidance is to determine when to check for a threat. Should a check be done for every valid return, when a return contributes to a line segment, or when a line segment is ended? We investigate this question and provide a workable initial approach.

2. Identification of a Collision Threat

The next step in the obstacle avoidance problem is the determination of the existence of a collision threat. This process begins with the declaration of what constitutes a collision threat and then the ability to recognize it from the sonar data at run

time. An important criterion in the identification process is present (and intended) motion of the AUV.

3. Collision Avoidance Actions

With successful identification of a collision threat the next issue becomes deciding what actions need to be taken and when they must be initiated. Such actions must take into account the current phase of the mission.

E. SUMMARY

This chapter summarizes the problems addressed by this thesis. Real-time sonar classification and run-time collision avoidance are critical parts of autonomous operations. To achieve these features many problems must be solved. After the initialization of the sonar systems, preprocessing of raw sonar data must be performed. Then the real-time sonar classification problem is addressed by line fitting, polyhedron building, and object classification. The problem of obstacle avoidance includes determining what is a collision threat, when to check for a collision threat, and how to respond to a collision threat.

IV. THEORETICAL DEVELOPMENT

A. INTRODUCTION

This chapter examines the real-time sonar classification problem in detail. The sonar classification process begins with the initialization of the sonar system, and then the gathering and preprocessing of raw sonar data. Range/bearing data is then fitted to line segments using parametric regression. The polyhedron-building algorithm then takes the line segments and combines them to form a polyhedron representation of the underwater objects that caused the sonar returns. Object classification is done based on the attributes of the polyhedron.

The collision avoidance problem is solved in two steps: first the ability to detect a collision threat, and second the ability to react in time to avoid the collision. The ability to react in a timely fashion results in the requirement of collision threat evaluation at a much higher frequency than object classification.

The final stage of the classification process is the representation of the classified objects for path planning purposes. A solution to the path planning problem is demonstrated in (Leonhardt 96) using circle representations of obstacles, that are the product of the sonar classification process.

B. SONAR DATA

1. Initialization of Sonars

The parameters of the sonar to be set at the initialization phase are, maximum range, receiver gain, azimuth change step size, transmitter power, mode (ST1000 only), and numbers of bins (ST1000 only). The initial settings are based on knowledge of the operating area. The initialization of the sonar system also requires the initialization of the serial port that is to be used for communications. The serial port used is "/dev/ttya" on

the Sun Voyager. The initialization that has to occur for the serial port every time the Voyager is rebooted is as follows:

- Become Super User to gain necessary permissions
- cd /opt/CDsts
- ./cdsoftcar -y /dev/ttya

2. Gathering of Raw Data

Sonar data can be collected using the AUV's ST725 scanning sonar or the ST1000 profiling sonar. The sonar used is based on which type of sonar search is chosen from the three types of sonar searches that can be conducted. The first sonar search is the Transit search. As its name implies, this search is conducted when the AUV is transiting between points. The Transit search is a back-and-forth scanning search between relative bearings 320 degrees and 040 degrees. The other two searches are complete 360 degree searches, used to conduct a thorough search of an unknown area. These two searches are the Sonar search where the sonar is scanned 360 degrees and, the Rotate search where the AUV is rotated 360 degrees.

3. ST725 Scanning Sonar

The ST725 sonar is primarily used for the Transit search. The raw data from the ST725 data is received as a 256 bit string representing 64 four bit values. Thresholding, filtering and smoothing techniques were evaluated on raw data to determine the best algorithm for preprocessing.

Our present algorithm employs a nearest-strong-return criterion, where a farther return on the same bearing has to be greater than one level higher to override a nearer return. This prevents a close weak target from being masked by a farther strong target. A thresholding limit of eight was set for the initial sea-water experiments at Moss Landing Harbor. This threshold is based on preliminary sonar testing results and likely needs to be evaluated on a case-by-case basis dependent on the local sonar environment. Previous returns are not used to filter spurious data, since a large distant target might mask a small

closer target. Instead spurious returns are identified and rejected at the parametric regression level. Depending on the range scale used, some of the initial bins are ignored as self noise. The number of bins ignored is a function of the relative bearing of the sonar in order to reduce "baffling" (i.e. self-occlusion) problems. The output of the sonar preprocessing algorithm is a single range/bearing data pair.

Figure 4.1 shows example raw sonar data from Moss Landing. Corresponding output of the sonar preprocessing algorithm is shown in Fig. 4.2, demonstrating the efficiency of the preprocessing algorithm. Sonar returns in the forward port quadrant are from a pier.

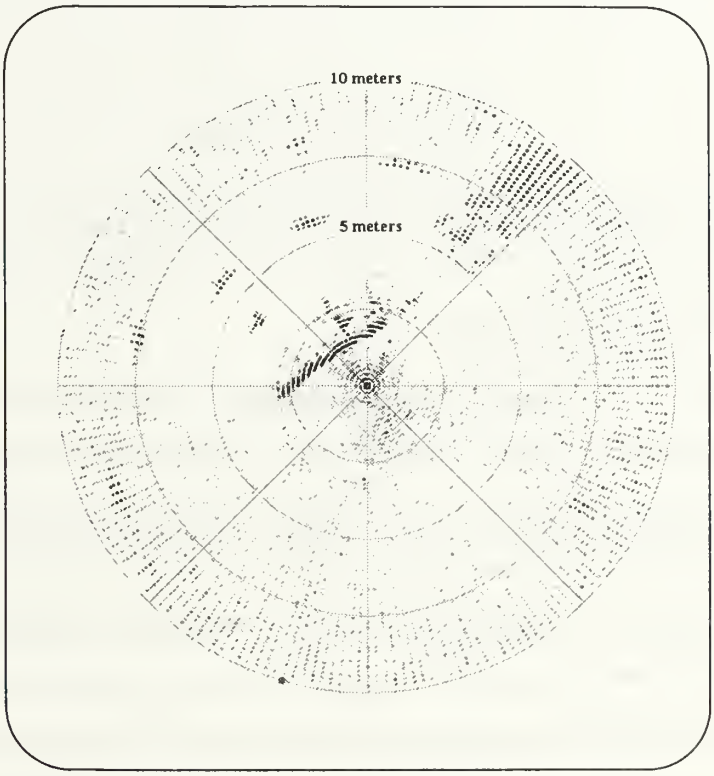

Figure 4.1. Raw Sonar Data Unprocessed. Returns in the Forward Port Quadrant are From a Pier.

4. ST1000 Profiling Sonar

The ST1000 sonar can operate both in a scanning mode where all of the preprocessing is the same as for the ST725, or in a profiling mode where the return is a

range in mm. Mode of operation is based on the initialization of the sonar head. The profiling mode provides more accuracy and requires less processing of the sonar data. Currently profiling mode is used during sector searches to take advantage of the superior range accuracy.

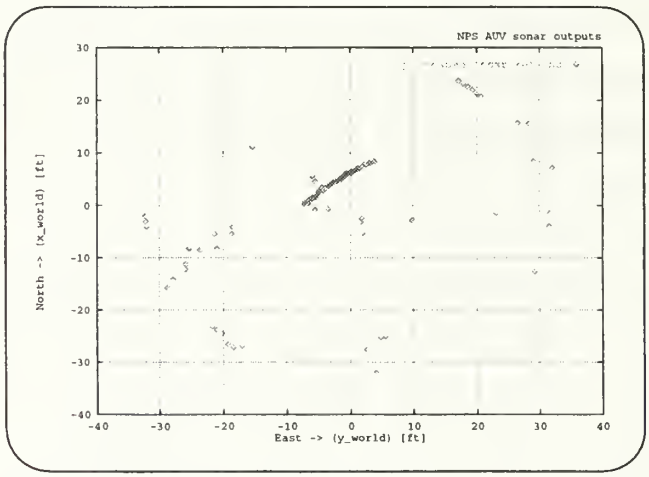

Figure 4.2. Preprocessed Sonar Data from Moss Landing
Using a Data Set Similar to that Shown in Fig. 4.1

5. Coordinate Transformation

The output from the sonar processing algorithm is a relative range and bearing. The range and bearing data is composed with the actual position and orientation of the Phoenix AUV, to determine x_return, and y_return, which are then used by the line fitting algorithm. The transformation to world coordinates is shown in Fig. 4.3.

Two-dimensional coordinate transformations are shown in Equations (4.1) and (4.2). The x and y values represent the center of buoyancy coordinates of the AUV, provided by the Execution level. The sonar offset from the AUV center is three feet. The translation to the actual sonar is done by the 3*sin(AUVheading) portion of the equation, with the units of feet.

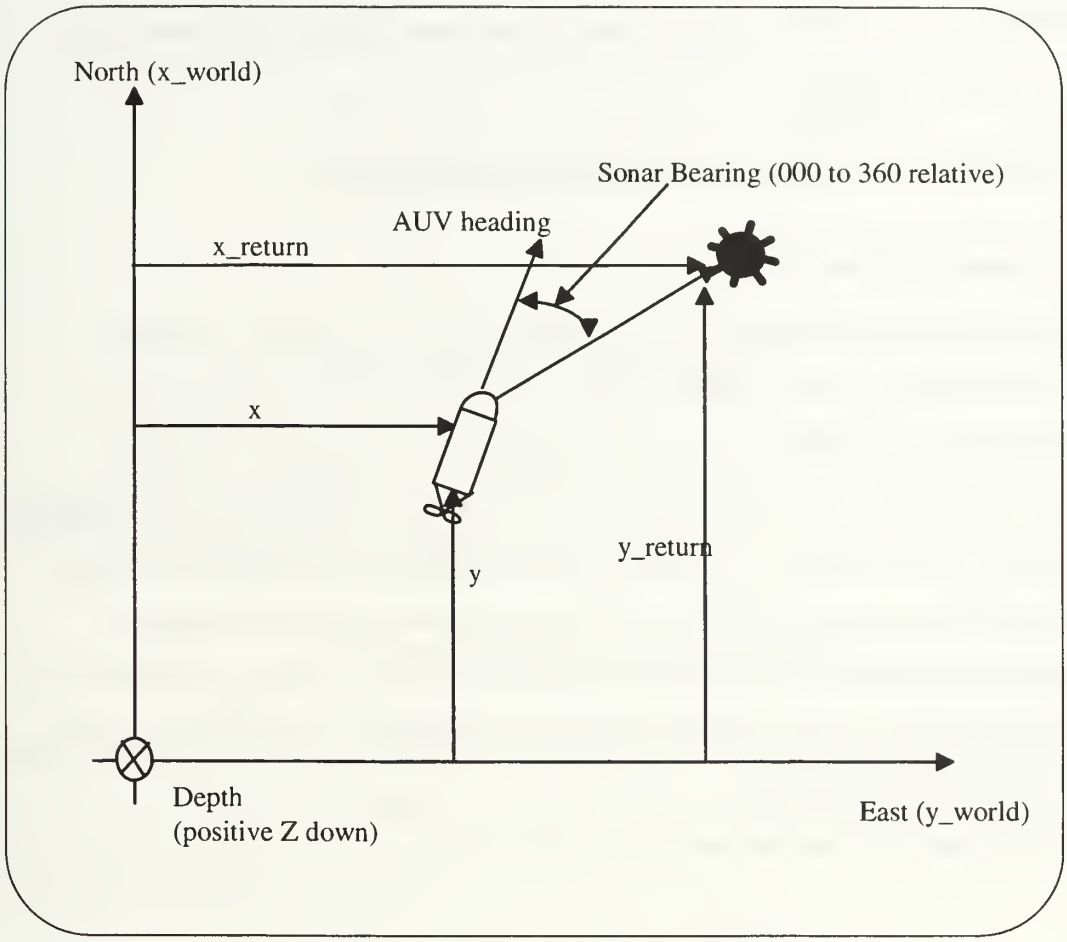

Figure 4.3. World Coordinate Transformation.

$$x\_return = x + 3 * \cos(AUVheading) + range * \cos(AUVheading + bearing) \qquad \text{(Eq. 4.1)}$$

$$y\_return = y + 3 * \sin(AUVheading) + range * \sin(AUVheading + bearing) \qquad \text{(Eq. 4.2)}$$

The effect of roll and pitch to the sonars is ordinarily small and is ignored due to the small errors of the AUV on these axes. The preprocessing of sonar data is independent of any motion by the Phoenix, due to the frequent (six to ten Hz) position updates provided by the execution level. The execution level dead reckons (estimates) the position using heading and speed. AUV speed is determined using a mathematical model at low speeds (less than one knot) and speed wheel sensor at higher speeds. The position is periodically

reset by the navigator module. The navigator module uses a kalman filter with inputs from DiveTracker and GPS.

C. LINE FITTING USING PARAMETRIC REGRESSION

1. Parametric Regression

The usual method of linear fitting is using a least-squares fitting algorithm in Cartesian coordinates. An unfortunate problem with this method is that it falls apart when data points are parallel to the y-axis, producing lines with infinite slope and resulting in a divide by zero situation (Kanayama 95). This problem has been eliminated by reformulation of least-squares line fitting using parametric representations of lines. The parametric approach is suited for real-time applications, due to its sequential incremental characteristics which can provide usable results at any time. A derivation of the algorithm follows. Further detail on the parametric regression line-fitting algorithm can be found in (Kanayama 95).

Given a set *R* of sonar data points:

$$R = \langle (x_i, y_i) | i = 1, ..., n \rangle \ . \qquad \text{(Eq 4.3)}$$

The moments of R are defined as

$$m_{jk} = \sum_{i=1}^{n} x^j y^k \ (0 \le j, k \le 2, and\ j + k \le 2) \qquad \text{(Eq 4.4)}$$

Notice that m_{00}= n. The centroid C of R is given by

$$C \equiv (\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}}) \equiv (\mu_x, \mu_y) \ . \qquad \text{(Eq 4.5)}$$

The secondary moments about the centroid are given by

$$M_{20} = \sum_{i=1}^{n} (x_i - \mu_x)^2 = m_{20} - (\frac{m_{10}}{m_{00}})^2 \qquad \text{(Eq 4.6)}$$

$$M_{11} = \sum_{i=1}^{n}(x_i - \mu_x)(y_i - \mu_y) = m_{11} - (\frac{m_{01}m_{10}}{m_{00}})^2 \qquad \text{(Eq 4.7)}$$

$$M_{02} = \sum_{i=1}^{n}(y_i - \mu_y)^2 = m_{02} - (\frac{m_{01}}{m_{00}})^2 \qquad \text{(Eq 4.8)}$$

The parametric representation of a line is adopted, with constants r and α. If a point p = (x, y) satisfies the equation

$$r = x\cos\alpha + y\sin\alpha \quad (-\pi/2 < \alpha < \pi/2) \qquad \text{(Eq. 4.9)}$$

then the point p is on a line L whose normal has an orientation α and whose distance from the origin is r shown in Fig. 4.4. In the parametric representation, the signed distance from a point $p_i = (x_i, y_i)$, to the line $L = (r, \alpha)$, is called the residual (δ), and calculated as follows.

$$\delta_i = x_i * \cos(\alpha) + y_i * \sin(\alpha) - r. \qquad \text{(Eq. 4.10)}$$

Therefore the sum of the squares of all of the residuals is

$$S = \sum_{i=1}^{n}(r - x_i\cos\alpha - y_i\sin\alpha)^2 . \qquad \text{(Eq 4.11)}$$

The best line fit of the set of data points will minimize S. The optimum line (r,α), must satisfy

$$\frac{\partial S}{\partial r} = \frac{\partial S}{\partial \alpha} = 0. \qquad \text{(Eq 4.12)}$$

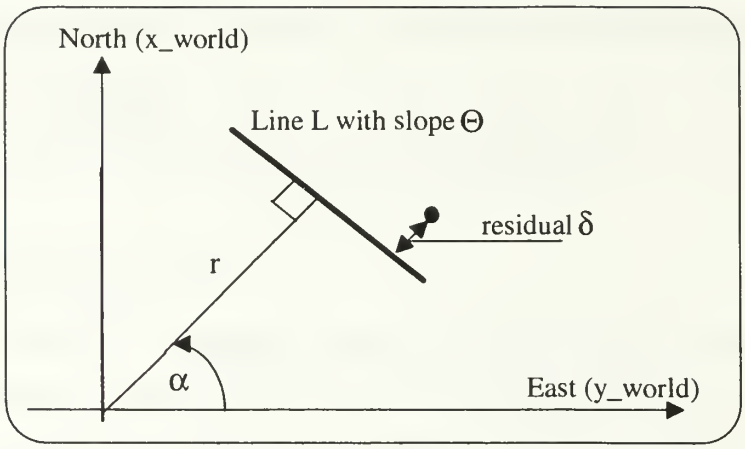

Figure 4.4. Representation of a Line. δ is the Residual from a Point.

Thus,

$$\frac{\partial S}{\partial r} = 2\sum_{i=1}^{n}(r - x_i\cos\alpha - y_i\sin\alpha)$$

$$= 2(r\sum_{i=1}^{n}1 - (\sum_{i=1}^{n}x_i)\cos\alpha - (\sum_{i=1}^{n}y_i)\sin\alpha) \qquad \text{(Eq 4.13)}$$

$$= 2(rm_{00} - m_{10}\cos\alpha - m_{01}\sin\alpha)$$

$$= 0$$

and

$$r = \frac{m_{10}}{m_{00}}\cos\alpha + \frac{m_{01}}{m_{00}}\sin\alpha = \mu_x\cos\alpha + \mu_y\sin\alpha \qquad \text{(Eq 4.14)}$$

where r may be negative. Substituting r in (Eq 4.11) by (Eq 4.14),

$$S = \sum_{i=1}^{n}((x_i - \mu_x)\cos\alpha + (y_i - \mu_y)\sin\alpha)^2. \qquad \text{(Eq 4.15)}$$

Finally,

$$\frac{\partial S}{\partial \alpha} = 2\sum_{i=1}^{n}((x_i - \mu_x)\cos\alpha + (y_i - \mu_y)\sin\alpha)(-(x_i - \mu_x)\sin\alpha + (y_i - \mu_y)\cos\alpha) \qquad \text{(Eq 4.16)}$$

$$= 2\sum_{i=1}^{n}((y_i-\mu_y)^2-(x_i-\mu_x)^2)\sin\alpha\cos\alpha +$$
$$2\sum_{i=1}^{n}(x_i-\mu_x)(y_i-\mu_y)(\cos^2\alpha-\sin^2\alpha)$$

$$= (M_{02}-M_{20})\sin 2\alpha + 2M_{11}\cos 2o$$

$= 0$ for a perfect line fit.

Therefore

$$\alpha = \frac{atan2(-2M_{11},M_{02}-M_{20})}{2} \qquad \text{(Eq 4.17)}$$

Here atan2 is the modified arctangent function which returns an angle in the proper quadrant. The solutions for the line parameters generated by a least-squares fit are given by Equations (4.14) and (4.17).

The equivalent ellipse of inertia for the original n points is an ellipse which has the same moments around the center of gravity. M_{major} and M_{minor} are moments about the major and minor axes respectively, shown in Fig. 4.5.

$$M_{major} = (M_{20}+M_{02})/2 - \sqrt{(M_{02}-M_{20})^2/4+M_{11}^2} \qquad \text{(Eq 4.18)}$$

$$M_{minor} = (M_{20}+M_{02})/2 + \sqrt{(M_{02}-M_{20})^2/4+M_{11}^2}. \qquad \text{(Eq 4.19)}$$

The diameters d_{major} on the major axis and d_{minor} on the minor axis of the equivalent ellipse are

$$d_{minor} = 4\sqrt{M_{major}/m_{00}} \qquad \text{(Eq 4.20)}$$

$$d_{major} = 4\sqrt{M_{minor}/m_{00}} \qquad \text{(Eq 4.21)}$$

We define ρ, the ellipse thinness ratio, to be the ratio of d_{minor} and d_{major} :

$$\rho = \frac{d_{minor}}{d_{major}} \qquad \text{(Eq 4.22)}$$

A small ρ (near zero) means a thin ellipse. As ρ increases toward 1, the ellipse opens to a circle representing a non-linear set of points. For this reason, ρ, is used as a testing parameter for linearity checks.

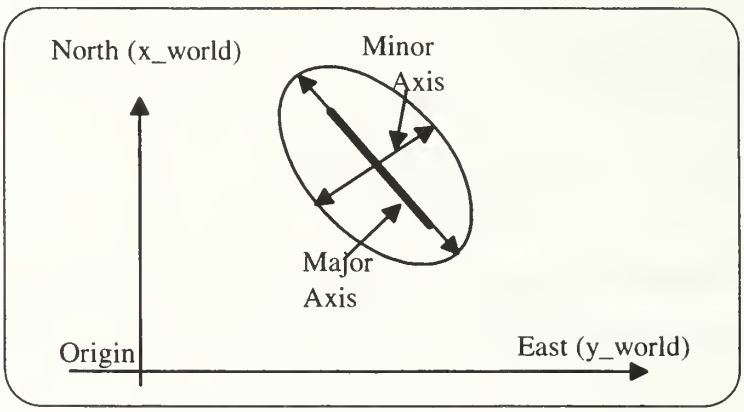

Figure 4.5. Equivalent Ellipse of Inertia.

2. Representation of Line Segments

The variables maintained during the building of a line segment are enumerated in Fig. 4.6. The sequential nature of the parametric regression algorithm is supported by consistently maintaining the moments and secondary moments. These summations are then updated every time a point is added to the line, keeping computational complexity O(1) rather than O(N) while adding points.

3. Starting Line Segments

The previously existing expert system required five data points to begin a line segment. A suitable set was found using a sliding window. This was the how the sonar module also started line segments in the early stages. However, once data was collected in an actual sea water environment, it was discovered that starting with four data points provided better results. This window is large enough that spurious returns will not start a

line segment, and small enough to detect far objects that will not produce that many returns.

The sliding window is implemented using two four-element arrays: initx and inity, as shown in Fig. 4.6. These arrays are filled with the four most recent valid sonar returns. A line is fitted to the data using Equations (4.14) and (4.17). If the thinness of the line is satisfactory (less than 0.1 for this implementation) then a segment is started. If the proposed line by the sliding window does not meet validity requirements the oldest point is thrown out and the next return is added. This process is repeated until the start of a valid line segment is found.

4. Building Line Segments

To reduce attempts to add the current sonar return to the current line segment when it does not fit, filtering of the sonar returns is performed. This filtering can often detect the end of a line segment without performing the computations necessary to include it into the current line and then ending it. The filtering consists of comparing the residual, Equation (4.10), to an maximum allowable distance from the line and comparing the residual to an maximum weighting of the standard deviation, σ which is calculated:

$$\sigma = \sqrt{M_{major}/(n-1)}. \qquad \text{(Eq. 4.23)}$$

The comparison is

$$\delta < \max(C1 * \sigma, C). \qquad \text{(Eq. 4.24)}$$

If the point is within these parameters it is then added to the line, and the thinness ratio is checked for the new line segment, if the line segment has exceeded the allowable thinness, the line segment is ended, and the current point is stored for the next line segment. If the line remains acceptable all of the line segment parameters are updated.

```
typedef  struct
  {
    double      theta;
    double      r;
    int         num_points;
    int         line_status;
    double      initx[4],inity[4];
    double      sgm_delta_sq;
    double      start_time;
    double      startx,starty;
    double      endx,endy;
    double      sgmx,sgmy;
    double      sgmx2,sgmy2;
    double      sgmxy;
    double      d_minor,d_major;
  } SEG_DAT;
```

Figure 4.6. Segment Building Data Structure.

5. Ending Line Segments

A line segment needs to be ended when the latest data point no longer forms an acceptable line, or when no sonar return is received for five seconds. Ending a line due to an unacceptable data point is determined during the preceding section on building line segments. A line segment is also be ended when the distance between the current return and the last return added to the line is not within an acceptable range (two feet for this implementation). When ending a line segment the final calculations are performed to end the line segment. When a line segment is ended the line parameters are calculated a final time and stored in a LINE_SEG structure. One parameter that is only calculated at the ending of a line segment is the orientation of the line, since the orientation is not used for building a line segment and only for the combination of line segments. The orientation Θ is calculated using the atan2 function, $\Theta = atan2(y\_end - y\_start, x\_end - x\_start)$. The need for consistency in the orientation calculation regardless of the scanning direction of the sonar can be seen in Fig. 4.7. Where the scan in the clockwise direction (Scan B)

would determine an orientation of 45 degrees, and the scan in the counter clockwise direction (Scan A) would calculate an orientation of -135 degrees.

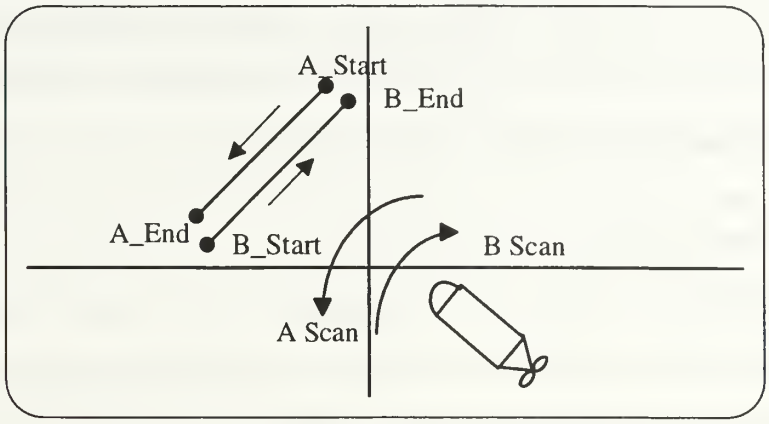

Figure 4.7. Scanning Direction and Line Orientation.

The scanning direction is not the only possible problem as shown in Fig. 4.8, AUVs scanning in the same direction still may not produce the same results with a simple atan2(endy - starty, endx - startx) calculation. For this example AUV scan "A" would produce an orientation equal to -135 degrees, while AUV scan "B" would calculate 45 degrees for the same segment, even though both AUVs are scanning in the clockwise direction.

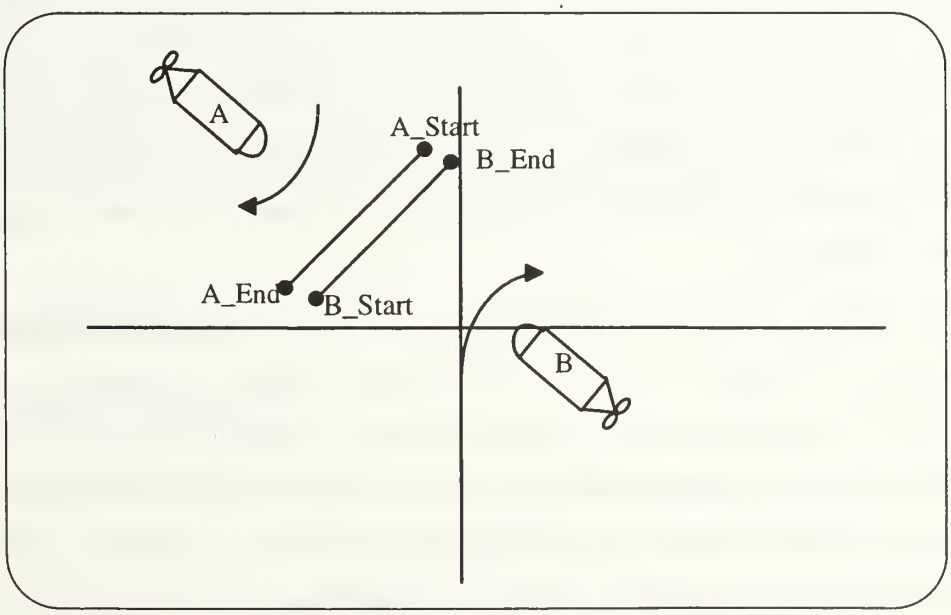

Figure 4.8. Orientation Calculation from Opposite Sides of a Line Segment.

To provide the necessary consistency for the orientation comparison, the α of the line segment is used. The issue to take into consideration when using this comparison is that the value of r may be negative. When adjacent segments have opposing r values, both r and α are negated for the second segment to permit proper comparison.

D. BUILDING OBJECTS FROM LINE SEGMENTS

1. Underwater Objects: Convex not Concave

Underwater objects of interest have predominantly convex shape. This fact is the basis for the adjacent line combination algorithm used. If an object does have concave features, it is instead represented as more than one object. Since concave objects are ordinarily not of concern, and since the algorithm always produces a valid world representation of convex and concave objects for path planning purposes, this is an acceptable approach.

2. Object Building

Segments are combined to form polyhedron representations of objects. The first condition checked to determine whether or not to combine segments is the distance between the line segments. The distance comparison is done between the end points of one line segment to the end points of the follow-on line segment, for both possibilities this allows for out-of-order combination. The comparison using the most current line segment's start point and the previous line segments end point is done first, as this is the most likely combination.

If the distance between the line segments is less than the permitted maximum tolerance the next comparison is orientation of the line segments. Alternatively normals may be used for comparison instead of line orientations. If the line segments are adjacent and colinear, then they are grouped together. The tolerance of the colinear check is relaxed the closer the line segments are to each other, this is due to the fact that the closer the line segments are the increased probability that they belong to the same object.

If the segments are not colinear it is then determined whether they are convex or concave. Due to the characteristics of underwater objects the segments are grouped together if they are convex and are not grouped together if they are concave. The problem of a consistent comparison between line segments occurs due to the possible change in sonar scanning direction, as well as AUV heading and bearing rate. A consistent comparison is needed that is independent of scan direction and relative position, as shown in Fig. 4.9.

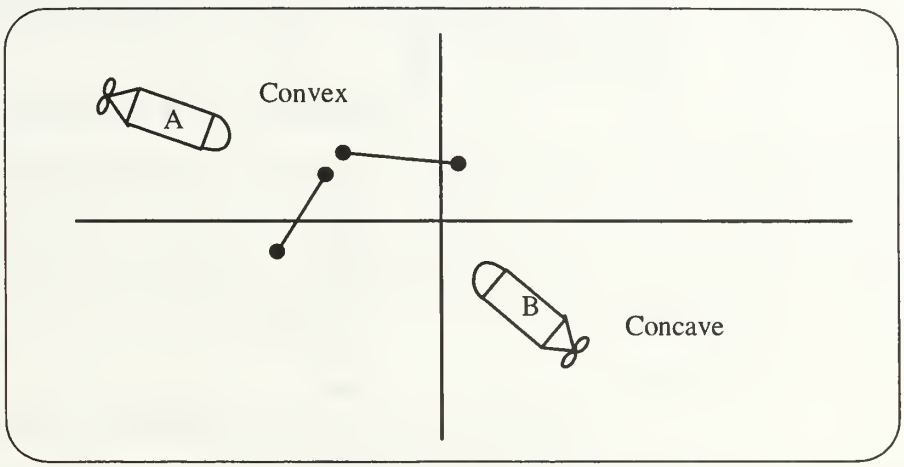

Figure 4.9. Convex versus Concave.

To solve this problem the bearings of the line segments relative to the AUV are calculated and compared to determine the line segment which is more clockwise. The clockwise line segment is then treated as the second line segment in the comparison, $\alpha_1 - \alpha_2$. If the result of that comparison is negative, the segments are convex with respect to the AUV.

E. CLASSIFICATION

1. Check If New Object

The initial part of classification is to ensure that the new object has not already been represented in the world. This is accomplished by comparing the classification of the objects and the centroids, if the classification is a wall. If the classification is a mine the test compares the areas of the objects as well as the centroids. The comparison is done

between the current object and all of the objects already defined in the world. More work is needed here. The first area to address would be to combine overlapping polyhedra. This can occur when the Phoenix AUV moves to another position relative to the target as demonstrated in Fig. 4.10. One way to combine polyhedra might be to calculate weighted averages of centroids and radii as shown in Fig. 4.11.

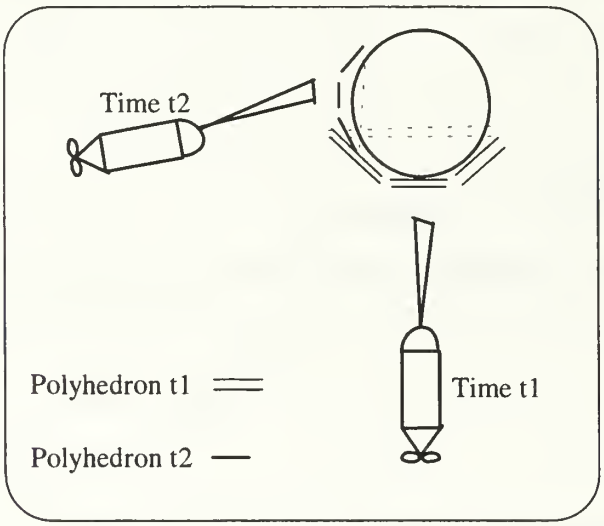

Figure 4.10 Overlapping Polyhedra.

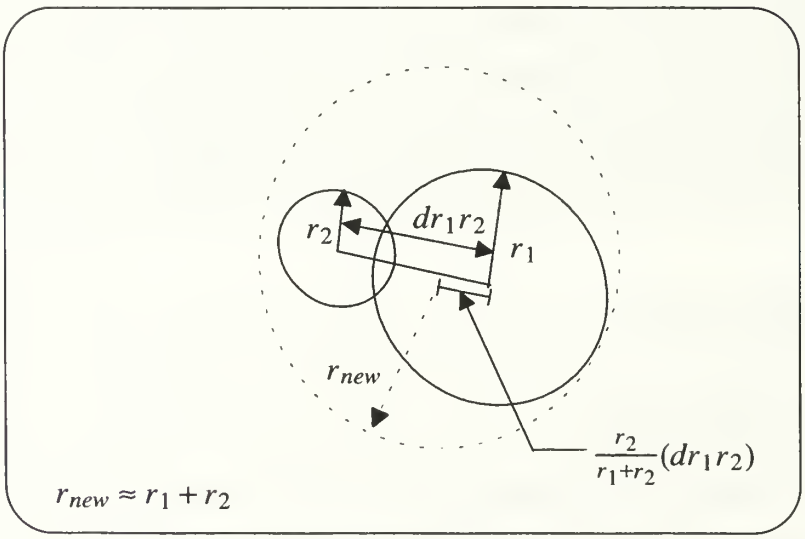

Figure 4.11. Merging Overlapping Circles.

2. Sequential Rule Firing

Classification is done based on the strength of returns, linearity and area of the polyhedra, as well as any known characteristics of the environment. All new polyhedra are tested to see if they are already represented in the world model. A check for a possible moving target looks for objects that are identical but have traveled at a finite speed. Much more work is possible here.

F. REPRESENTATION OF CLASSIFIED OBJECTS

1. Method of Representation

Representation of the objects is achieved using circle representations for path planning purposes. This representation is valid as most objects in the underwater environment can be adequately approximated by individual cylinders or walls of cylinders.

2. Representation of Linear Objects (Walls)

The Replanner module uses circle representation of objects for path planning. Since the replanner uses circles, linear objects need to be represented as circles. A predetermined radius is used to create circle representations of the linear object. This is done as shown in Fig. 4.12.

```
/* split a wall into circles of global_radius */
   number_of_circles = ceiling(length/global_radius);
   delta_x = (tailx - headx) / number_of_circles;
   delta_y = (taily - heady) / number_of_circles;
   for (i = 0; i < number_of_circles; ++i)
     {
        fprintf(new_circle_ptr,"Circle %6.4f %6.4f %6.4f %6.4f\n",
           (headx + i*delta_x),(heady + i*delta_y), z ,global_radius);
     }
```

Figure 4.12. Circle Representation of Linear Objects.

3. Representation of Polyhedra

The area of a polyhedron is the summation of the triangle areas shown in Fig. 4.13. The area of a single planar triangle is given by Equation (4.25).

$$Area_{\Delta} = \frac{1}{2}|(X_2 - X_1)(Y_3 - Y_1) - (X_3 - X_1)(Y_2 - Y_1) \quad \text{(Eq 4.25)}$$

The polyhedron is represented using centroid_x and centroid_y shown in Equations (4.26) and (4.27). The radius of the circle is calculated in Equation (4.28).

$$Centroid\_x = \frac{\Sigma(start\_x + end\_x)}{2 * number\_of\_line\_segme} \quad \text{(Eq 4.26)}$$

$$Centroid\_y = \frac{\Sigma(start\_y + end\_y)}{2 * number\_of\_line\_segme} \quad \text{(Eq 4.27)}$$

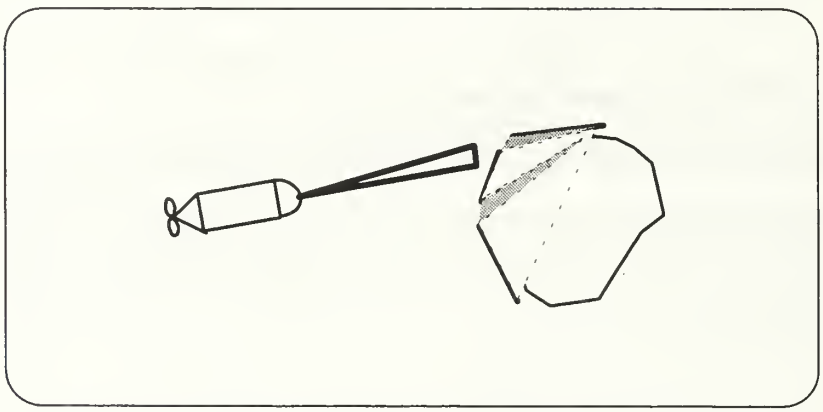

Figure 4.13. Summing Triangle Areas to Determine Polyhedra Area.

$$radius = \sqrt{2 * Area_{polyhedron}/\pi} \quad \text{(Eq 4.28)}$$

The area is multiplied by two for a safety range from the polyhedron.

G. COLLISION THREATS

1. When to Check?

The issue of collision avoidance is an important safety issue of the AUV. To ensure the safety of the AUV the frequency of checks must be often enough to guarantee a collision threat will not be missed. The elimination of the unnecessary processing of spurious returns is also an issue of concern. Both problems are handled by the collision threat check, which is performed for every sonar return that contributes to a line segment. If a return does not contribute to a line segment then the return is considered spurious. Further work will be needed to discriminate between spurious returns and objects which do not provide consistent returns.

2. What is a Collision Threat?

A collision threat is any object that lies in the path of the AUV within a five foot range i.e., a little more than one-half ship length. The safe width that is required for the AUV is four feet. With the required four foot width for passage and five foot safety range, the relative bearings that are checked for collision threats are from 336 degrees to 024 degrees, as shown in Fig. 4.14. This simple check will detect most problems when transiting between waypoints.

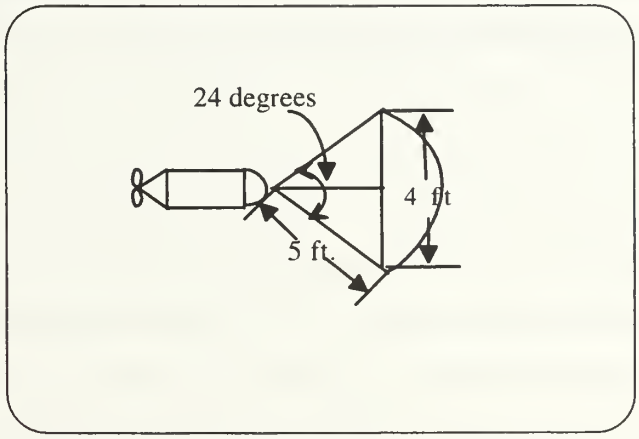

Figure 4.14. Collision Avoidance Safety Range.

3. Collision Avoidance Actions

When a collision threat is detected the sonar module passes the message "COLLISION_THREAT" to the OOD module. The OOD module then orders a full backing bell until all headway is removed from the AUV. Future implementations should include a "collision warning" message, when an object is in the path of the AUV, but not close enough to be a collision threat, since this warning would allow for less drastic measures and easier recovery. Other actions will need to be developed for hovering mode. The best approach is probably to stop, hover in place, back away as necessary to avoid collision, map the new collision threat, and replan the path.

H. SUMMARY

The algorithms above have been implemented in the current sonar module operating on the Phoenix. The sonar module initializes the sonar system at startup. The run-time processing of the module includes the gathering, processing and transformation of the raw sonar data. The data is then fitted to line segments using parametric regression. The implementation of the object building and classification algorithms have demonstrated the ability to provide the required data for the autonomous operation of an underwater vehicle. Collision avoidance implementation has also been supported with message passing to the OOD when a collision threat occurs.

V. EXPERIMENTAL DESIGN AND RESULTS

A. INTRODUCTION

The goal for the Phoenix was to conduct a successful sea water mission of detecting, localizing and classifying a mine-like object. Sonar code developments were first tested in the virtual world, then the test tank and finally a larger mission demonstrating the AUV's capabilities.

B. VIRTUAL WORLD TESTING

1. Using the Virtual World

A virtual world has been used throughout the development of this code (Brutzman 94). The virtual world allows the user to run all vehicle software verbatim, testing interprocess communications and algorithm correctness. While the virtual world allows for testing correctness, it does not test for hardware robustness, and is currently somewhat of an ideal environment even with sensor errors inserted.

2. Experiments

The initial virtual world testing was a simple mission with the AUV in the center of the test tank. A 360 degree rotation was performed to gather sonar data and test the classification algorithms. The resulting raw sonar data is shown in Fig. 5.1. The line segments formed from the sonar data and the circle representations of those line segments are shown in Fig. 5.2. The test was done once with no knowledge of the "world." The initial run produced an output file called "new_world," this file is the representation of the environment computed by the program. The second test uses the output from the first run as input for the "world." This was to test the correctness of the "check_if_new" function. The ability to compare objects with the known environment reduces the communication between the OOD and the Sonar module, as well as an unnecessary collision threat reports. The results of the testing were satisfactory.

3. Test results

Figure 5.1 shows the virtual world range/bearing data. Figure 5.2 shows the line segments fitted to the data and the center of the circles produced for path planning purposes.

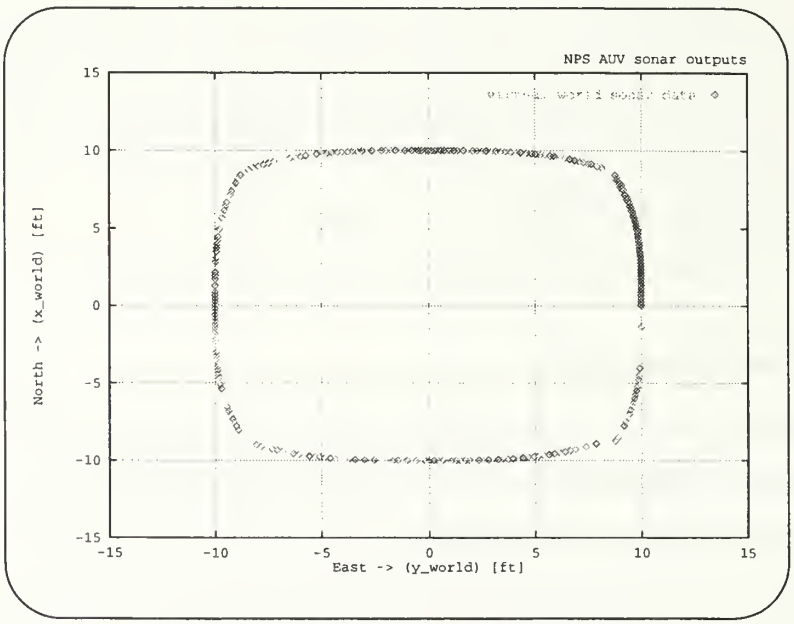

Figure 5.1 Sonar Data from Virtual World.

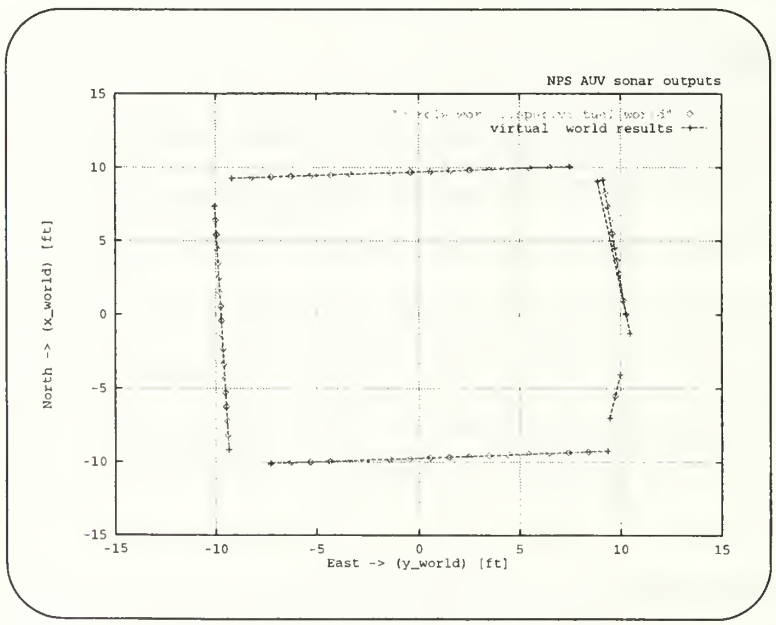

Figure 5.2 Fitted Line Segments and Centers of Circle Representations.

C. TANK TESTING

1. Preprocessing Real Sonar Data

None of the computer hardware used in this thesis was installed or connected prior to this work. The test tank was a useful environment at first to allow for communication between the Voyager computer and the sonar systems. The test tank was the first opportunity to evaluate raw sonar data and determine the best methods for preprocessing. The sonar data collected was processed through various thresholding, smoothing, and filtering algorithms. The final algorithm developed for the sonar preprocessing is represented in Fig. 5.3. The input to the algorithm is a 64-bin range array, with each bin containing a number between zero and 15 representing the average strength of the return over that portion of the range scale. The first step of the algorithm is bin thresholding, which is the process of ignoring some of the initial bins to eliminate interference from self noise. The number of bins that are ignored completely is based on the relative bearing of the sonar, in order to eliminate false returns from the AUV itself. The next step is the sequential evaluation of all of the remaining bins and testing the bin value against a threshold value. A threshold value of seven was used. If the bin strength is greater than the threshold value, it is a candidate return. The next step is to compare it to the current maximum value. The comparison between bin values is done by weighting the closer bins such that a distant strong contact will not obscure a closer weaker contact (which may present a collision threat).

2. Position Problems

The main disadvantage of the tank testing is the lack of positioning data available to the Phoenix. Without an accurate dead reckon position for a moving vehicle, sonar data is useless since a bearing and range mean nothing without a point of origin. Given these limitations and separate problems with the dead-reckon model, the only useful sonar testing that was accomplished in the test tank was performed with the Phoenix stationary and in a known position.

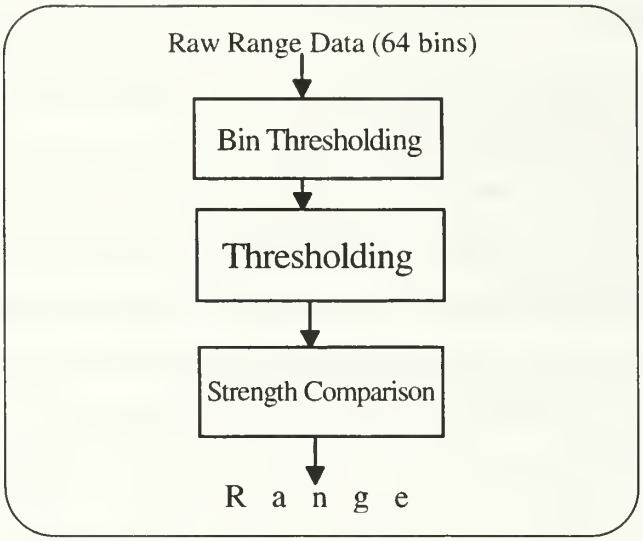

Figure 5.3. Sonar Preprocessing Algorithm.

3. Testing Results

Tank tests conducted a 360 degree scan, and the data was processed by the sonar module, with the results being evaluated for correctness. These tests were performed with a mine-like object in the tank, to evaluate the classification rules. The plots and output files demonstrate the classification of a mine-like object, in the test tank using a stationary AUV. Sonar detection, localization and classification results were satisfactory as demonstrated by Fig 5.4 and Fig. 5.5. The preprocessed sonar data and the fitted line segments are shown in Fig. 5.4. The objects created by the module are shown in Fig. 5.5, with the mine-like object classified at coordinate (4.06, 0.25) with a radius of one foot.

D. SEA WATER TESTING

1. Moss Landing Harbor

The first ever sea-water testing of the Phoenix took place at Moss Landing Harbor in January 1996. Many problems were discovered during this testing. The initial tests allowed for the evaluation of many systems that cannot be tested in the tank i.e., GPS and DiveTracker. Unfortunately the positioning data of the Phoenix was not as accurate as

needed to accomplish the initial transit/search/transit mission designed for the harbor. Poor dead reckoning and hardware reliability problems produced many unusable results.

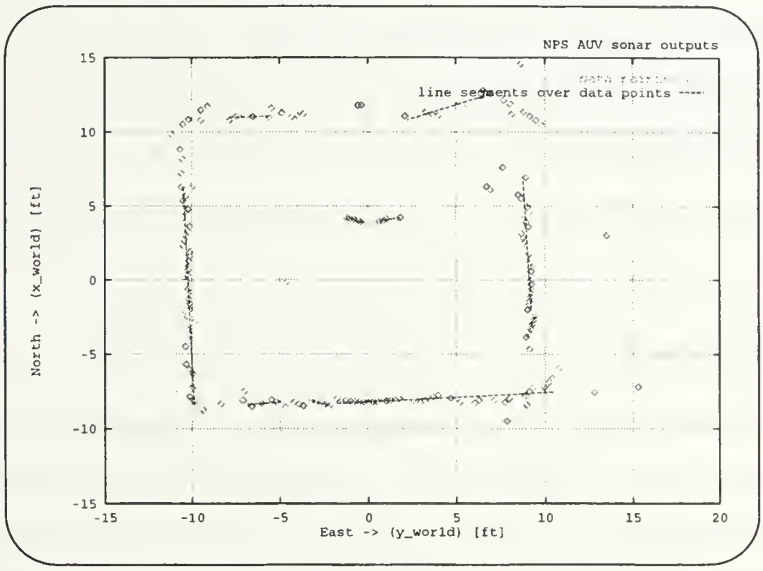

Figure 5.4. Sonar Data and Fitted Lines from Tank testing.

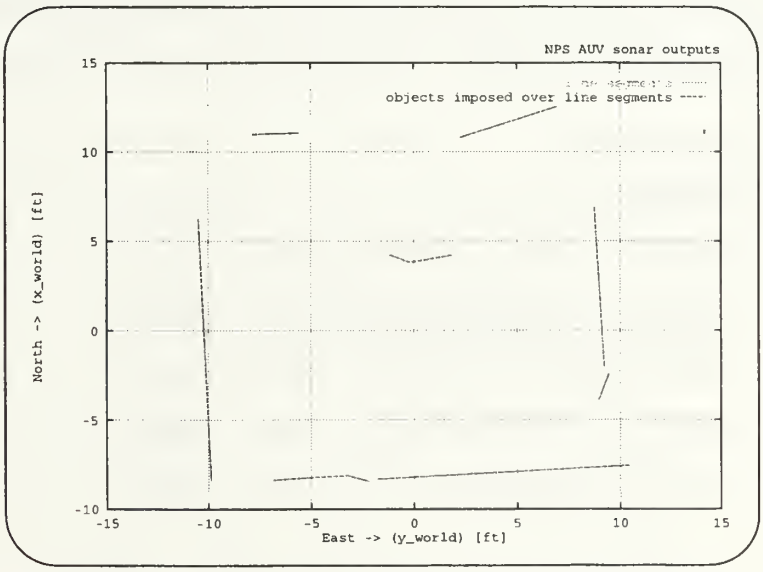

Figure 5.5. Objects from Tank Testing. Note Small Mine-Like Object at (4.0,0.0).

2. Real World Situations

The sonar data gathered from stationary and moving scans were used to test the algorithms in a true sea-water environment. The major improvement to the algorithm that was produced by this testing was the line ending condition of two consecutive zero returns. Prior to the harbor testing all sonar testing was done in man-made environments. Enclosed conditions produced the anomaly of always receiving a valid sonar return, and therefore the condition of no return was not discovered until Moss Landing. This was an excellent result.

3. Data and Results

The data gathered during the two weeks of testing was not as useful as originally expected. This was due to the lack of accurate position data while the Phoenix was transiting. Stationary data was gathered and the results were shown in Figs. 4.1 and 4.2. The output circle_world.input file is shown in Fig. 5.6 demonstrating the format of the input to the replanner module. Object radii equal to 1.0000 indicates that these circles were generated to approximate a wall.

# OBJECT	X	Y	Z	Radius
Circle	9.6522	13.0779	2.0000	1.0000
Circle	10.4248	13.6633	2.0000	1.0000
Circle	11.1974	14.2488	2.0000	1.0000
Circle	11.9700	14.8342	2.0000	1.0000
Circle	12.7426	15.4197	2.0000	1.0000

Figure 5.6. Circle World Input File from Moss Landing Data.

E. POOL TESTING

1. NPS Pool

After the results of the Moss Landing testing were evaluated we decided to attempt further testing in the swimming pool at NPS. The goal was again a mission of detecting, localizing and classifying a mine-like object. The lack of accurate position information was once again the pitfall.

A very successful mission was accomplished during the pool testing. The Phoenix was placed in a known location and a sonar search was conducted while the Phoenix was stationary. This experiment demonstrated the ability for all software components to be running together and achieve real-time performance from the sonar module. The pre-processed sonar data and fitted line segments from this experiment are shown in Fig. 5.7. The centers of the circles produced for path planning are plotted in Fig. 5.8. The mine-like object was detected at coordinate (30,33-36), although at this range with the ST725 the object only produced four returns and was classified as "unknown". The ability to demonstrate the real-time capabilities of this module was an excellent result.

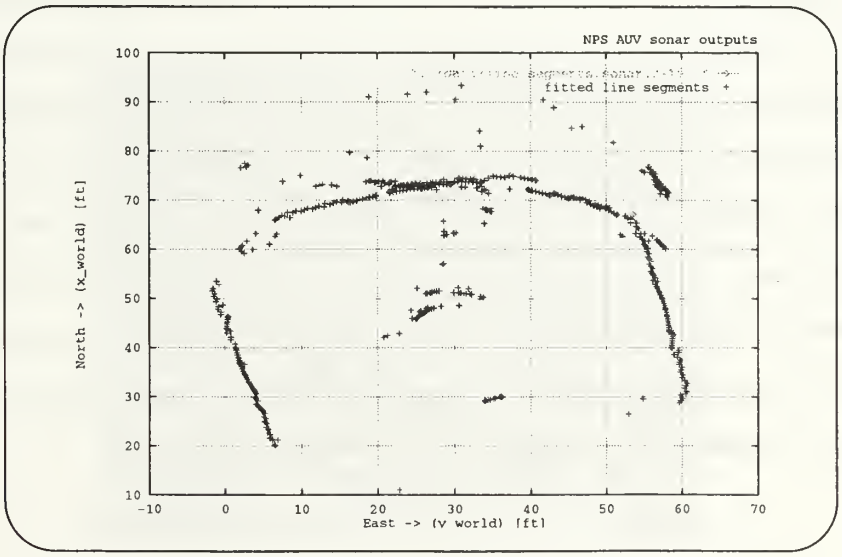

Figure 5.7. Processed Sonar Data and Fitted Line Segments.

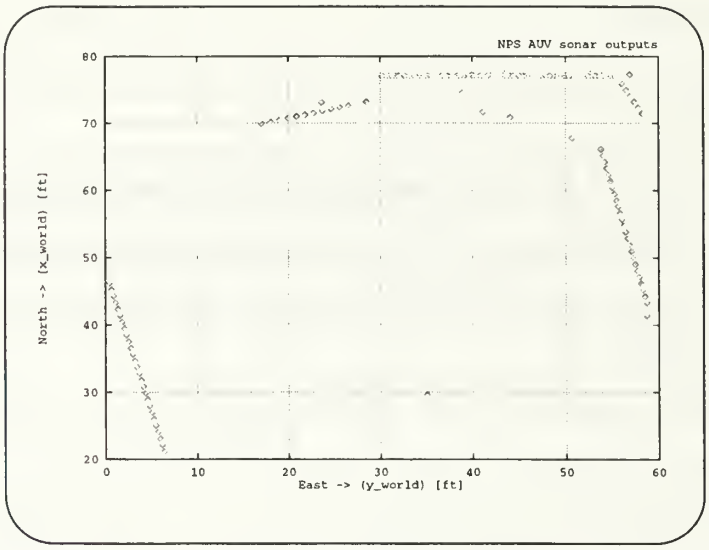

Figure 5.8. Circle Representation of the Sonar World. Note Small Mine-Like Object at (30,35).

F. FOLLOW-ON TESTING

1. New Virtual World

The data that was gathered from the Moss Landing testing was implemented into the virtual world. The virtual world was also updated to provide sonar data based on the bearing of the sonar head, provided by the sonar module, and the graphical representation of objects in the virtual world. A computational geometric sonar model provided returns accurate within inches, with approximately a five percent error rate in generated returns. This new version of the virtual world was used to perform sonar classification and path replanning tests. Having a complete geometric model and complete real-time visualization of sonar bearings and ranges immediately clarified several difficulties, enabling immediate correction of several long-standing problems. This was merely one of many occasions where visualization improvements resulted in suprisingly profound insights which were previously elusive.

2. Results

The results of this testing were excellent, as an end-to-end mission of detecting, localizing, classifying and replanning around a mine-like object was accomplished. A picture of the mission running is shown in Fig. 5.9. The output from the sonar module is shown in Fig. 5.10 displaying the sonar representation of the walls and the mine-like object. The raw sonar data is shown in Fig. 5.11. The circle world output from the mission is plotted in Fig. 5.12.

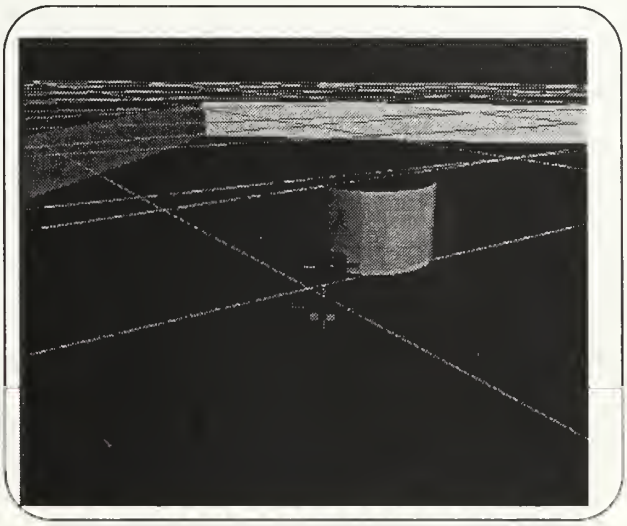

Figure 5.9. Virtual World Mission Snapshot.

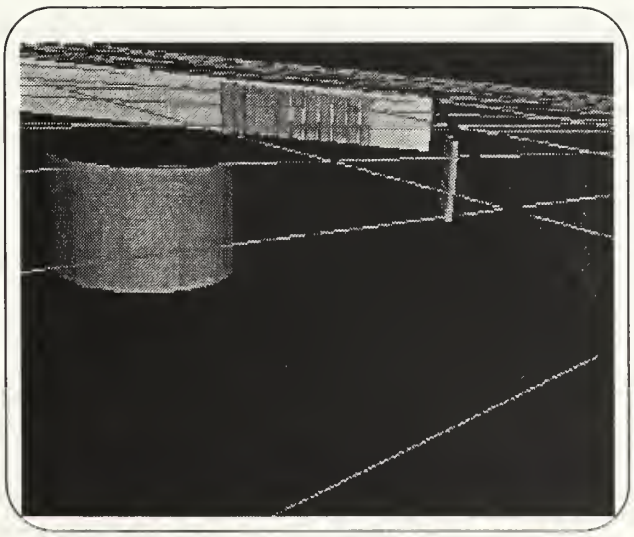

Figure 5.10. The Sonar Module's Circle World from Virtual World Data. Cylinders Shown Here Represent Classified Objects from Fig. 5.12.

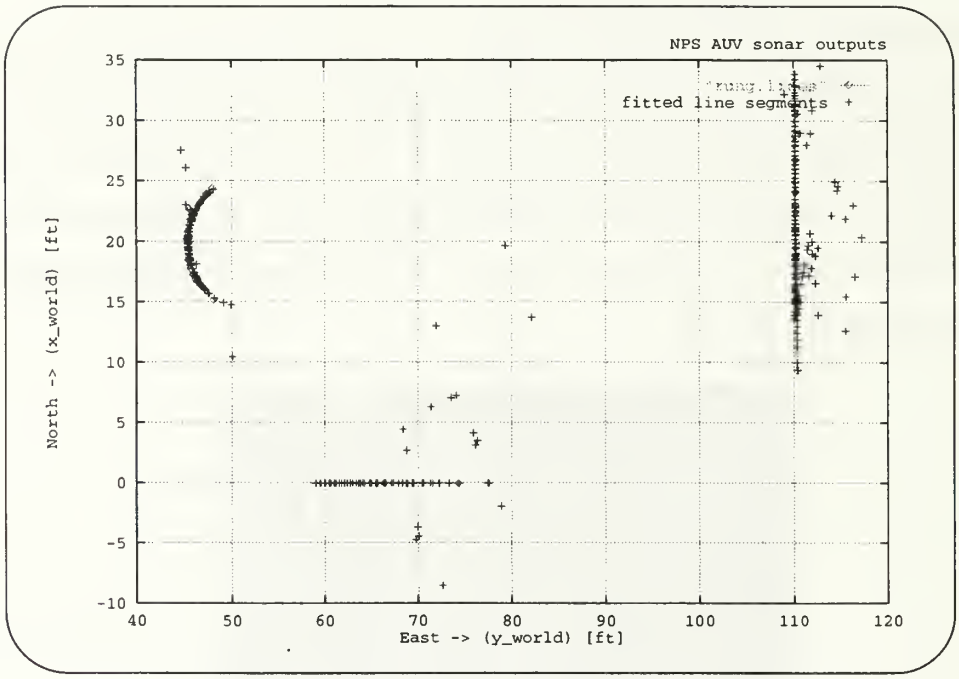

Figure 5.11. Raw Sonar Data with Fitted Lines from Virtual World Mission.

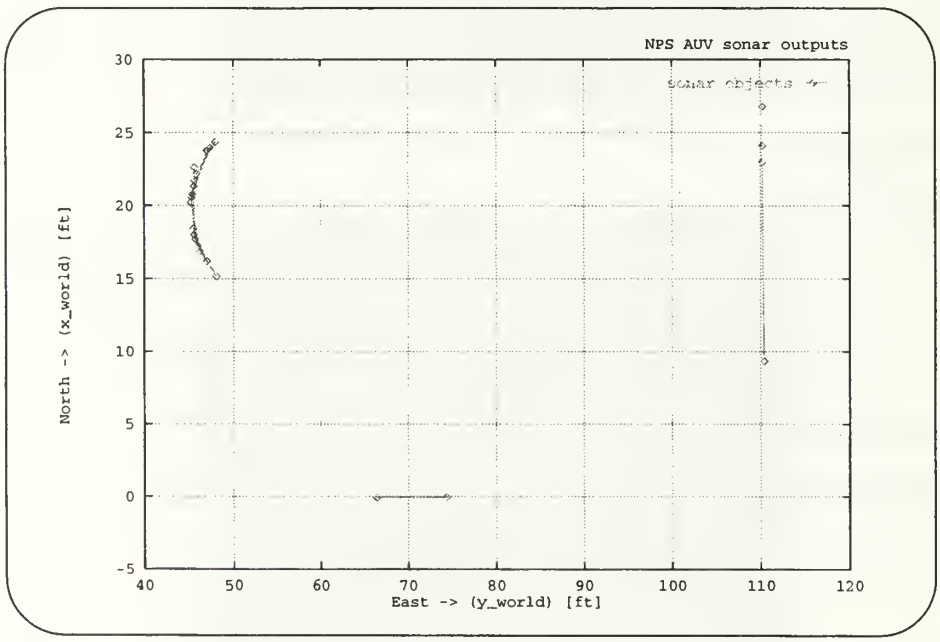

Figure 5.12. Classified Objects Created from Virtual World Mission. Note the Curved Object Centered at Coordinate (20,46) Was Classified Mine-Like.

G. SUMMARY

Testing was a four step process. The initial testing done in the virtual world allowed for the testing of the sonar module without the overhead of deploying the vehicle. This was used to test and refine the basic algorithms. The virtual world also supplied the ability to establish the communications between all of the parts of the software architecture. The next step of tank testing, was useful in the refinement of the hardware interface portion of the sonar module, but as mentioned above the lack of accurate position data limited the amount of testing possible. The next testing, accomplished in sea water, provided useful insights to the ability of the sonar and the algorithms. Many hardware failures and less-than-expected accuracy of position data limited the amount of useful results gathered from the sonar system during the Moss Landing tests. The fourth set of tests were performed using a greatly enhanced geometric sonar model in simulation. Accurate position data, accurate returns and some noisy returns were successfully analyzed and classified using vehicle hardware and vehicle software in real time.

The many difficulties involved with the deployment of the Phoenix, e.g., hardware failures and logistic support, demonstrate what an invaluable asset a virtual world is in the development of software. Despite the disappointing shortfalls of end-to-end system testing, enough successful tests were conducted using the sonar module aboard Phoenix to conclude that real-time sonar classification is achievable. We believe we have a working system now. Further in-water testing is needed to tune coefficient choices and validate overall system performance in a variety of real-world situations.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

1. Real-Time Classification

The sonar module developed by this work has shown the ability to process sonar data in real time. The real time classification of objects was not accomplished in an untethered waterborne mission, due to many hardware and software problems, although subsequent testing produced real-time classification in the virtual world. The dead reckon position data of the Phoenix is not currently accurate or consistent enough to support the real-time classification of sonar objects when underway. The sonar module does process the sonar data received during waterborne testing and produces the required outputs in real-time. Without reliable position information this data gathered cannot be verified. Nevertheless the correctness of the algorithm has been demonstrated using sonar data gathered with a fixed position and a 360 degree sonar search conducted using vehicle hardware. With these two positive results, it is evident that real-time object classification is achievable. Further improvements in dead reckoning are likely and corresponding sonar results will be reevaluated.

2. Collision Avoidance

A simple collision avoidance algorithm has been implemented in the Phoenix. The results of this algorithm are independent of the actual position of the Phoenix. The algorithm uses relative sonar range and bearing for the determination of a collision threat. This simplicity is a desirable feature, since it will protect the vehicle regardless of the navigational accuracy. More testing and additional development will no doubt further improve collision avoidance capabilities.

3. Object Representation

The representation of classified objects, for the purposes of path planning, is performed with circle representations as shown in Fig. 5.3. This data is shared with the replanner module by creating a file, which is later used by the replanner module as the input for a path planning process. Circle representations are adequate for most (if not all) target obstacles encountered by an AUV.

B. RECOMMENDATIONS FOR FUTURE WORK

1. Testing

The current Tactical level needs to be further tested with adequate dead reckon position information. Performing a complete mission will demonstrate the capabilities and/or improvements needed for all current software.

2. VxWorks

The need for a shared memory system is evident by the large amount of message passing required by the current implementation. The shared-memory needs of the tactical level combined with the real-time requirements of the execution level can both be satisfied with the implementation of the VxWorks operating system. It is likely that performance gains are possible using shared memory. Process profiling analysis is needed first before embarking on a system reconfiguration. Regardless of whether such a transition is made, current results show that shared memory is not required and a standard Unix approach can work.

3. Video Camera Correlation with Sonar

The next logical step in the MCM efforts of AUVs is to use a camera to provide visual support of the classification performed by the sonar module. The idea here is that once a mine-like object is classified, the AUV can transit to a closer location and acquire visual confirmation, or provide the new classification of the object. Image processing

will remain independent from sonar classification, and is not needed for mine-like object classification or safe path planning.

4. Expanding Classification Rules

The current implementation has demonstrated the ability to process sonar data in real time and create line segments from that data. The ability to build and classify objects has also been accomplished, but the need for more classification rules is evident. Now that the real time problem of sonar classification has been solved, the next improvement is the expansion of the rules for detailed classification. We want to be able to discriminate between mines, rocks, fish and other moving entities. We also want to combine "blobbed" data points which come from the same target but do not yet provide adequate resolution for line fitting.

5. Improved Collision Avoidance Reactions

The need for improved collision avoidance actions is obvious. The current implementation is a fail-safe method suitable for the current testing. Improvements will be needed once the platform is ready for more complex testing. The improvements can be made both at the OOD level, by taking less severe actions in accordance with the phase of the mission, and at the sonar module level by creating another message (e.g. "collision_warning") that might occur at a farther range from fast-moving objects to provide the OOD more time to react to the situation. Another improvement that can easily be made is for the OOD module to replan after avoiding the collision, instead of aborting the mission which the current implementation does.

6. Virtual World Sonar Model

The virtual world provides sonar data for objects defined in the virtual world, by the user. The next step in the progression of the virtual world is the implementation of a realistic noise distribution for the sonar data, in order to present data that is comparable to data collect in waterborne experiments. The current sonar data produced by the virtual

world is an excellent representation of the ST1000 sonar, as it returns only a range and bearing. An implementation that returns a 33 byte data string similar to the ST725 (or the ST1000 in scanning mode) could be very useful in further testing of raw sonar pre-processing. This implementation would require valid sonar returns on the same bearing for multiple targets and would provide for the testing of pre-processing algorithms. This would help improve the ability to locate weaker closer contacts that can be masked by farther stronger contacts.

7. ST1000 Implementation

Due to the difficulties experienced with the waterborne testing, the ST1000 was never fully implemented as an available sonar to the sonar module. The code for communications with the ST1000 is already written. The testing required deals mostly with the processing of the returns in the profiling mode. Operating the ST1000 in the scanning mode would work identically to the ST725. The implementation of the ST1000 will also require three-dimensional transformations. These transformations will be required due to the fact that the ST1000's one degree conical beam will be more sensitive to the AUV's pitch and roll than the 24 degree vertical beam of the ST725. This is a straightforward task for implementation.

C. SUMMARY

This work resulted in a fully implemented sonar module for the Phoenix. Improvements were made to the previous algorithm with the addition of checks for loss of sonar returns and proximity checks between returns. Further improvements were made in the polyhedra building with the algorithm being modified to allow for combination of segments that are produced in any scanning direction. Although the entire mission of detecting, localizing and classifying a mine-like object was not quite accomplished due to other problems, a major step was taken with the demonstration of the sonar module's ability to produce real-time in-water results detecting and localizing a mine-like object.

Subsequent testing in the virtual world demonstrated convincing real-time detection, localization and classification of a mine-like object.

LIST OF REFERENCES

Boorda, J. M., "Mine Countermeasures - An Integral Part Of Our Strategy And Our Forces," White Paper, December 1995.

Brutzman, Don, *A Virtual World for an Autonomous Undersea Vehicle*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, December 1994. Available at *http://www.stl.nps.navy.com/~brutzman/dissertation*

Brutzman, Don, *NPS AUV Integrated Simulator*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1992.

Brutzman, Don, "Virtual World Visualization for an Autonomous Underwater Vehicle," *Proceedings of the IEEE Oceanic Engineering Society Conference OCEANS 95*, San Diego California, October 12-15 1995, pp. 1592-1600. Available at *ftp://taurus.cs.nps.navy.mil/pub/auv/oceans95.ps.Z*

Brutzman, Don, *Software Reference: A Virtual World for the NPS Autonomous Underwater Vehicle (AUV)*. Available at *http://www.stl.nps.navy.mil/~auv/*

Brutzman, Don, Compton, Mark A. and Kanayama, Yutaka, "Autonomous Sonar Classification using Expert Systems," *Proceedings of the IEEE Oceanic Engineering Society Conference OCEANS 92*, Newport, Rhode Island, October 26-29 1992, pp. 554-559. Available at *ftp://taurus.cs.nps.navy.mil/pub/auv/oceans92.ps.Z*

Burns, Michael, *An Experimental Evaluation and Modification of Simulator-based Vehicle Control Software for the Phoenix Autonomous Underwater Vehicle (AUV)*, Master's Thesis, Naval Postgraduate School, Monterey, California, April 1996. Available at *http:/www.cs.nps.navy.mil/research/auv*

Byrnes, R.B., *The Rational Behavior Model: A Multi-Paradigm, Tri-level Software Architecture for the Control of Autonomous Vehicles*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, March 1993.

Kanayama, Yutaka J., *CS4313: Lecture Notes Addendum: Generalized Least Squares Fitting*, Naval Postgraduate School, Monterey, California, April 1995.

Kanayama, Yutaka J., Kovalchik, Joseph G., Chuang, Chien-Liang, Kelbe, Frank E., "Motion Planning for Autonomous Mobile Robots," *Proceedings of the Autonomous Vehicles in Mine Countermeasures Symposium*, Monterey, California, April 4-7 1995.

Leonhardt, Bradley, *Mission Planning and Mission Control Software for the Phoenix AUV: Implementation and Experimental Study*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1996. Available at *http:/www.cs.nps.navy.mil/research/auv*

Marco, D. B., "Autonomous Underwater Vehicle: Hybrid Control of Mission and Motion," *Journal of Autonomous Robots*, 1996.

McClarin, Dave, *Discrete Asynchronous Kalman-Filtering of Navigation Data for the Phoenix Autonomous Vehicle,* Master's Thesis, Naval Postgraduate School, Monterey, California, March 1996.

NASA Software Technology Branch, *CLIPS Reference Manual*, Lyndon B. Johnson Space Center, Houston, Texas, 1991.

Scrivener, Art, *Acoustic Underwater Navigation of the Phoenix Autonomous Underwater Vehicle using the DiveTracker System*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1996.

Tritech International Ltd, *ST-725 Sonar: User Manual*, Mike E. Chapman Company, Duvall, Washington, 1992.

Tritech International Ltd, *ST-1000 Sonar: User Manual*, Mike E. Chapman Company, Duvall, Washington, 1992.

Tritech International Ltd, *ST-Sonar Head Control: Technical Notes*, Mike E. Chapman Company, Duvall, Washington, 1992.

APPENDIX A. SOURCE CODE FOR SONAR MODULE

```
/*************************************************************************
   FILENAME:       sonar.c

   AUTHOR:         Mike Campbell
   DATE:           15 March 1996

   PURPOSE:        Gathers all of the sonar data and performs real time
                   object classification to support mine-hunting and supply
                   run-time collision avoidance.

   REVISION:       This code is constantly being improved and expanded current
                   revision is available at:
                   http://www.stl.nps.navy.mil/~auv/tactical/

   FUNCTIONS:      sonar()
                   linear_fitting()
                   start_segment()
                   add_to_line()
                   add_circle()
                   normalize()
                   normal2()
                   normal()
                   struct LINE_SEG *end_segment()
                   reset_accumulators()
                   build_poly()
                   print_list()
                   classify_poly()
                   Power()
                   quadrant()
                   triangle_area()
                   init_next_poly()
                   end_poly()
*************************************************************************/
/* #include "vxWorks.h"  */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <string.h>
#include <stddef.h>
#include "sonar_globals.h"
#include "../execution/globals.h"
#include "../execution/defines.h"
#include "../execution/statevector.h"
#define RECORD_SIZE sizeof(struct LINE_SEG)
#define MAXLINE 132

void linear_fitting();
void start_segment();
void add_to_line();
void add_circle();
double normaliz();
```

```
double normal();
struct LINE_SEG *end_segment();
void reset_accumulators();
void build_poly();
void print_list();
void classify_poly();
double Power();
int quadrant();
double triangle_area();
void init_next_poly();
void end_poly();
double normal2();

char   Sonar_String_back[MAXBUFFERSIZE];
char   Sonar_data[MAXBUFFERSIZE];
char   Sonar_Read_to_clear[MAXBUFFERSIZE];
extern int  Sonar_to_OOD_fd[2],OOD_to_Sonar_fd[2];
extern int  Sonar_telemetry_fd[2];
extern int  initialize_sonar_systems();
extern double  Ping_Sonar();

/*************************************************************************
   FUNCTION:      sonar()

   AUTHOR:        Mike Campbell

   DATE:          4 March 1996

   PURPOSE:       Handles the communications within and outside of this
                  module

   RETURNS:       none, sends sonar data through socket comms to calling
                  function. (tactical.c)
*************************************************************************/
void sonar(void)

{

 /* Open file, testing for success */

   if ((outfile = fopen ("data_points.sonar", "w")) ==((FILE *) 0))
   {
      printf("Error opening out file \n");
      exit(0);
   }
   if ((out2 = fopen ("line_segments.sonar", "w")) == ((FILE *) 0))
   {
      printf("Error opening out file \n");
      exit(0);
   }
   if ((objectfile = fopen ("objects.sonar", "w")) == ((FILE *) 0))
   {
      printf("Error opening out file \n");
      exit(0);
```

```
}
if ((new_world = fopen("new_world.sonar", "w")) == ((FILE *) 0))
  {
   printf("Error opening new world file. \n");

  }
if ((worldfile = fopen("world", "r")) == ((FILE *) 0))
  {
   printf("World file does not exist. \n");
  }
else
  {
    while (fgets(line, MAXLINE, worldfile))
   {
   sscanf(line,"%f %f %f %f %f %f %f %f %f  %i %i %i",
   &t1,&t2,&t3,&t4,&t5,&t6,&t7,&t8,&t9,
   &i1,&i2,&i3);
   world[poly_num].start          = t1;
   world[poly_num].end            = t2;
   world[poly_num].head_of_poly   = NULL;
   world[poly_num].headx          = t3;
   world[poly_num].heady          = t4;
   world[poly_num].tailx          = t5;
   world[poly_num].taily          = t6;
   world[poly_num].centroidx      = t7;
   world[poly_num].centroidy      = t8;
   world[poly_num].area           = t9;
   world[poly_num].seg_count      = i1;
   world[poly_num].status         = 13;
   world[poly_num].classification= i3;
   ++poly_num;
    }
   }
segment_data.theta = 0;
segment_data.r = 0;
segment_data.num_points = 0;
segment_data.line_status = 0;
while (1)
{
 if (read(Sonar_telemetry_fd[0],Sonar_data,MAXBUFFERSIZE)== -1)
   {}
 else
   {
    parse_telemetry_string(Sonar_data);
    if (read(OOD_to_Sonar_fd[0],Sonar_data,MAXBUFFERSIZE)== -1){}
    else
    {

     if (strcmp (Sonar_data,"INITIALIZE") == 0)
      {
       if (LOCATIONLAB)
        {
          sprintf(Sonar_String_back,"SONAR_INITIALIZED");
          Sonar_mode = 0;
```

```
      SONAR_725_bearing = 0;
      Scan_direction = -1;
      write(Sonar_to_OOD_fd[1],Sonar_String_back,MAXBUFFERSIZE);
    }
   else
    {
     st_path = initialize_sonar_systems();
     if (st_path > 0)
      {
       sprintf(Sonar_String_back,"SONAR_INITIALIZED");
       Sonar_mode = 0;
       SONAR_725_bearing = 0;
       Scan_direction = -1;
       write(Sonar_to_OOD_fd[1],Sonar_String_back,MAXBUFFERSIZE);
       printf("%s\n",Sonar_String_back);
      }
     else
     {
      sprintf(Sonar_String_back,"SONAR_INITIALIZED_FAILED");
      write(Sonar_to_OOD_fd[1],Sonar_String_back,MAXBUFFERSIZE);

     }
    }
  }
 else if (strcmp (Sonar_data,"SONAR_SEARCH") == 0)
   {
    Sonar_mode = 1;

   }
 else if (strcmp (Sonar_data,"ROTATE_SEARCH") == 0)
   {
    Sonar_mode = 2;

   }
 else if (strcmp (Sonar_data,"QUIT") == 0)
  {
   classify_poly(poly_num);
   print_list();
   fclose(textfile);
   fclose(outfile);
   fclose(out2);
   fclose(worldfile);
   fclose(new_world);
   exit(0);
  }
}
switch(Sonar_mode)
 {
  case -1:
      break;
  case 0:
    if ((Scan_direction == -1) && (SONAR_725_bearing >= 60.0) &&
                                    (SONAR_725_bearing <= 300.0))
        Scan_direction = 1;
```

```
  else if ((Scan_direction == 1) && (SONAR_725_bearing <= 300.0) &&
                                        (SONAR_725_bearing >= 60.0))
       Scan_direction = -1;
  if (SONAR_725_bearing > 170 && SONAR_725_bearing < 190)
    bin_threshold = 12;
  else
    bin_threshold = 4;
  if (LOCATIONLAB)
  {
   SONAR_725_range = AUV_ST725_range;
  }
else
  SONAR_725_range =
                    Ping_Sonar(st_path,Scan_direction,bin_threshold);
  SONAR_725_bearing =
         normal2(SONAR_725_bearing + step_size *-0.9*Scan_direction);
  break;
case 1:
   if (Search_status == 0)
     {
      if (!LOCATIONLAB)
       center_sonar(st_path);
      x_search = x;
      y_search = y;
      SONAR_725_bearing = 0.0;
      Search_status = 1;
     }
  if (SONAR_725_bearing > 170 && SONAR_725_bearing < 190)
    bin_threshold =12;
  else
    bin_threshold = 6;
   if (LOCATIONLAB)
    SONAR_725_range = AUV_ST725_range;
   else
    SONAR_725_range = Ping_Sonar(st_path,-1,bin_threshold);
   SONAR_725_bearing +=  step_size *0.9;
   if (SONAR_725_bearing > 360)
    {
      SONAR_725_bearing = normal2(SONAR_725_bearing);
      Search_status = 0;
      Sonar_mode = 0;
      sprintf(Sonar_String_back,"SONAR_SEARCH_COMPLETE");
      write(Sonar_to_OOD_fd[1],Sonar_String_back,MAXBUFFERSIZE);
    }
   break;
case 2:
  if (Rotate_status == 0)
     {
      if (!LOCATIONLAB)
       center_sonar(st_path);
      SONAR_725_bearing = 0.0;
      Rotate_status = 1;
      Rotate_count = 0;
     }
```

```
      if (LOCATIONLAB)
       SONAR_725_range = AUV_ST725_range;
      else
       SONAR_725_range = Ping_Sonar(st_path,0,8);
      Rotate_count += 0.5;
      if (Rotate_count > 200)
       {
         Rotate_status = 0;
         Sonar_mode = 0;
         sprintf(Sonar_String_back,"ROTATE_SEARCH_COMPLETE");
         write(Sonar_to_OOD_fd[1],Sonar_String_back,MAXBUFFERSIZE);
        }
     break;
   }
if (SONAR_725_range >= 0.1)
 {
  one_bad_range = 0;
  if (Sonar_mode != 1 && t > 1.0 && !LOCATIONLAB)
  {
   x_return = x + 3*cos(normal(psi*M_PI/180)) +
             cos(normal(psi*M_PI/180+
                        SONAR_725_bearing*M_PI/180))*SONAR_725_range;
   y_return = y + 3*sin(normal(psi*M_PI/180)) +
        sin(normal(psi*M_PI/180+
                        SONAR_725_bearing * M_PI/180))*SONAR_725_range;
  }
  else if (Sonar_mode == 1 && t > 1.0 && !LOCATIONLAB)
  {
   x_return = x_search + 3*cos(normal(psi*M_PI/180)) +
                cos(normal(psi*M_PI/180+
                        SONAR_725_bearing*M_PI/180))*SONAR_725_range;
   y_return = y_search + 3*sin(normal(psi*M_PI/180)) +
                 sin(normal(psi*M_PI/180+
                        SONAR_725_bearing*M_PI/180))*SONAR_725_range;
  }
  else if (t > 1.0)
  {
   x_return = x + 3*cos(normal(psi*M_PI/180)) +
                cos(normal(psi*M_PI/180+
                        AUV_ST725_bearing*M_PI/180))*SONAR_725_range;
   y_return = y + 3*sin(normal(psi*M_PI/180)) +
             sin(normal(psi*M_PI/180+
                        AUV_ST725_bearing*M_PI/180))*SONAR_725_range;
  }
  if (t >= 1.0 && (((x_return - old_x)*(x_return - old_x)+
                     (y_return - old_y)*(y_return - old_y)) >= 0.05))
   {
     sprintf(Sonar_String_back,"SONAR_725 %lf %lf %lf",
                SONAR_725_bearing ,SONAR_725_range,SONAR_725_strength);
     write(Sonar_to_OOD_fd[1],Sonar_String_back,MAXBUFFERSIZE);
     fprintf(outfile,"%lf %lf %lf \n",x_return ,y_return
                                          ,AUV_ST725_bearing);
     fflush (outfile);
     old_x = x_return;
```

```
            old_y = y_return;
            linear_fitting();
           }
          else
           {
            sprintf(Sonar_String_back,"SONAR_725 %lf %lf %lf",
                        SONAR_725_bearing, SONAR_725_range,SONAR_725_strength);
            write(Sonar_to_OOD_fd[1],Sonar_String_back,MAXBUFFERSIZE);
           }
          SONAR_725_range = 0.0;
         }
       else if (one_bad_return == 1 && segment_data.line_status > 1)
        {
          build_poly(end_segment,poly_num);
          reset_accumulators();
        }
       else
         one_bad_return = 1;
     }
   }
}

/*************************************************************************
   FUNCTION:       Power()

   AUTHOR:         Mike Campbell

   DATE:           4 March 1996

   PURPOSE:        Provides a function that raises numbers to powers.

   RETURNS:        A double of Base raised to the Exp.
*************************************************************************/

double Power(Base,Exp)

double Base;
int Exp;

{

   int Loop = Exp;
   double Total = 1.0;
   double BaseNum = Base;

   while(Loop)
   {
     if(Loop > 0){
       Total = Total * BaseNum;
      Loop--;
      }
     else
     {
       Total = Total / BaseNum;
```

```
        Loop++;
       }
    }
    return Total;
}



/*************************************************************************
    FUNCTION:       quadrant()

    AUTHOR:         Mike Campbell

    DATE:           4 March 1996

    PURPOSE:        Returns the quadrant of the angle as a number from 0 to 3,
                with 0 representing the +Y and -X quadrant and rotating CW
                from there.

    RETURNS:        An integer between 0 and 3.
*************************************************************************/
int quadrant(Angle)

double Angle;

{
  if(Angle > M_PI/2)
    return 0;
  else if(Angle > 0)
    return 1;
  else if(Angle > -M_PI/2)
    return 2;
  else
    return 3;
}

/*************************************************************************
    FUNCTION:       triangle_area()

    AUTHOR:         Mike Campbell

    DATE:           4 March 1996

    PURPOSE:        Calculates the area of a triangle represented by the points
                    X1,Y1 X2,Y2 and X3,Y3.

    RETURNS:        A double representing the area of the triangle P1P2P3.
*************************************************************************/

double triangle_area(X1,Y1,X2,Y2,X3,Y3)

double X1,Y1,X2,Y2,X3,Y3;
```

```
{
   double Ans;

   Ans = (0.5*(((X2 - X1)*(Y3 - Y1))-((X3 - X1)*(Y2-Y1))));
   return Ans;
}



/*************************************************************************
   FUNCTION:      normalize()

   AUTHOR:        Mike Campbell

   DATE:          4 March 1996

   PURPOSE:       Accepts a double representation of an angle (in radians)
             and returns an angle between -PI and +PI.

   RETURNS:       A double between -PI and +PI.
*************************************************************************/
double normalize(theta)

double theta;
{
  double Ans;

  if (theta < -M_PI)
   {
    Ans = theta + 2*M_PI;}
  else if (theta >= M_PI)
   {
    Ans = theta - 2*M_PI;}
  else
    Ans = theta;
  return Ans;
}
/*************************************************************************
   FUNCTION:      normal()

   AUTHOR:        Mike Campbell

   DATE:          4 March 1996

   PURPOSE:       Accepts a double representation of an angle (in radians)
                  and returns an angle between 0 and 2PI.

   RETURNS:       A double between 0 and 2PI.
*************************************************************************/
double normal(theta)

double theta;
{
```

```
  double Ans;

  if (theta < 0)
    Ans = theta + 2*M_PI;
  else if (theta >= 2*M_PI)
    Ans = theta - 2*M_PI;
  else
    Ans = theta;
  return Ans;
}

/*************************************************************************
   FUNCTION:      normal2()

   AUTHOR:        Mike Campbell

   DATE:          4 March 1996

   PURPOSE:       Normalizes numbers (in degrees) between 0 and 360.

   RETURNS:       A double between 0 and 360 degrees.
*************************************************************************/
double normal2(theta)

double theta;
{
  double Ans;

  if (theta < 0.0)
    Ans = theta + 360.0;
  else if (theta >= 360.0)
    Ans = theta - 360.0;
  else
    Ans = theta;
  return Ans;
}

/*************************************************************************
   FUNCTION:      linear_fitting()

   AUTHOR:        Mike Campbell

   DATE:          4 March 1996

   PURPOSE: This procedure controls the fitting of range data to straight
            line segments. First it collects four data points and establishes
            a line segment with it's interim data values. After the segment
            is established, the procedure tests each subsequent data point
            to determine if it falls within acceptable bounds before calling
            the least squares routine to include the data point in the line
            segment. After inclusion of the data point the segment is again
            tested to ensure the entire set of data points are linear enough.
            If any of the tests fail, the line segment is ended and a new one
            started. The completed line segment is stored in a data structure
```

```
            called segment, and segments are linked together in a linked list.

   RETURNS:        none, sends sonar data through socket comms to calling
                   function. (tactical.c)
*************************************************************************/

void linear_fitting()

{
   int num_points, line_status;
   double theta, r, sigma, delta,del_y, del_x;
   struct LINE_SEG *finished_segment;

   theta    = segment_data.theta;
   r        = segment_data.r;
   num_points     = segment_data.num_points;
   line_status      = segment_data.line_status;
   del_x = x_return - segment_data.endx;
   del_y = y_return - segment_data.endy;

/* FIRST CHECK TO SEE IF NEW POINT IS TO FAR FROM LAST POINT TO INCLUDE */

   if(num_points > 0)
    {
      if (del_x*del_x + del_y*del_y > 4.0)
       {
        if (line_status > 1)
          {
            finished_segment = end_segment();
            build_poly(finished_segment,poly_num);
          }
        reset_accumulators();
        segment_data.num_points = 0;
        segment_data.line_status = 0;
        line_status = 0;
        num_points = 0;
        }
     }
   if (line_status == 0) /* not enough data points yet */
     {
      segment_data.initx[num_points] = x_return;
      segment_data.inity[num_points] = y_return;
      segment_data.endx = x_return;
      segment_data.endy = y_return;
      segment_data.num_points += 1;
      if (num_points == 3)
        {
        start_segment();
        segment_data.line_status = 1;
        }
     }
   else
     {
      sigma = segment_data.sgm_delta_sq /(double) num_points;
```

```
delta = x_return * cos(theta) + y_return * sin(theta) - r;
if (fabs(delta) < residual_tolerance)||
      fabs(delta) < (sigma * sigma_weighting)
 {
 switch (line_status)
   {
   case 1:
     segment_data.num_points += 1;
     add_to_line(x_return, y_return);
     if (segment_data.line_status == 1)
        segment_data.line_status = 2;
     else if (segment_data.line_status == 5)
         {
           reset_accumulators();
           segment_data.initx[0] = segment_data.initx[1];
           segment_data.inity[0] = segment_data.inity[1];
           segment_data.initx[1] = segment_data.initx[2];
           segment_data.inity[1] = segment_data.inity[2];
           segment_data.initx[2] = segment_data.initx[3];
           segment_data.inity[2] = segment_data.inity[3];
           segment_data.initx[3] = x_return;
           segment_data.inity[3] = y_return;
           segment_data.num_points = 4;
           start_segment();
           segment_data.line_status = 1;
         }
       break;
   case 2:
     segment_data.num_points += 1;
     add_to_line(x_return, y_return);
     if (segment_data.line_status == 5)
       {
        finished_segment = end_segment();
        build_poly(finished_segment,poly_num);
        reset_accumulators();
        segment_data.initx[0] = x_return;
        segment_data.inity[0] = y_return;
        segment_data.num_points = 1;
        segment_data.line_status = 0;
        segment_data.endx=x_return;
        segment_data.endy=y_return;
       }
       break;
   case 3:
     segment_data.initx[1] = x_return;
     segment_data.inity[1] = y_return;
     segment_data.line_status = 4;
     break;
   case 4:
     segment_data.num_points += 1;
     add_to_line(segment_data.initx[1], segment_data.inity[1]);
     if (segment_data.line_status == 5)
      {
       finished_segment = end_segment();
```

```
      build_poly(finished_segment,poly_num);
      reset_accumulators();
      segment_data.initx[2] = x_return;
      segment_data.inity[2] = y_return;
      segment_data.num_points = 3;
      segment_data.line_status = 0;
      segment_data.endx=x_return;
      segment_data.endy=y_return;
     }
     else
      {
       segment_data.num_points += 1;
       add_to_line(x_return, y_return);
       if (segment_data.line_status == 5)
       {
        finished_segment = end_segment();
        build_poly(finished_segment,poly_num);
        reset_accumulators();
        segment_data.initx[0] = x_return;
        segment_data.inity[0] = y_return;
        segment_data.num_points = 1;
        segment_data.endx=x_return;
        segment_data.endy=y_return;
        segment_data.line_status = 0;
       }
     else
       segment_data.line_status = 2;
    }
    break;
   }
 }
else
 {
  switch (line_status)
   {
    case 1:
    case 2:
     segment_data.initx[0] = x_return;
     segment_data.inity[0] = y_return;
     segment_data.line_status = 3;
     break;
    case 3:
     finished_segment = end_segment();
     build_poly(finished_segment,poly_num);
     reset_accumulators();
     segment_data.initx[1] = x_return;
     segment_data.inity[1] = y_return;
     segment_data.num_points = 2;
     segment_data.endx=x_return;
     segment_data.endy=y_return;
     segment_data.line_status = 0;
     break;
    case 4:
     finished_segment = end_segment();
```

```
            build_poly(finished_segment,poly_num);
            reset_accumulators();
            segment_data.initx[2] = x_return;
            segment_data.inity[2] = y_return;
            segment_data.num_points = 3;
            segment_data.endx=x_return;
            segment_data.endy=y_return;
            segment_data.line_status = 0;
            break;
          }
       }
    }
}


/*************************************************************************
   FUNCTION:       start_segment()

   AUTHOR:         Mike Campbell

   DATE:           4 March 1996

   PURPOSE:        This procedure establishes a new line segment with the four
                   data points contained in segment_data.init(x and y). It
                   writes the appropriate data to the interim values in
                   segment_data.

   RETURNS:
*************************************************************************/

void start_segment()

{
    double theta, r, mux, muy, muxx, muyy, muxy,sds = 0;
    int i,j;

    segment_data.start_time = t;
    segment_data.startx = segment_data.initx[0];
    segment_data.starty = segment_data.inity[0];
    segment_data.endx   = segment_data.initx[3];
    segment_data.endy   = segment_data.inity[3];
    for (i = 0; i < 4; ++i)
       {
      segment_data.sgmx  += segment_data.initx[i];
      segment_data.sgmy  += segment_data.inity[i];
      segment_data.sgmx2 += Power(segment_data.initx[i],2);
      segment_data.sgmy2 += Power(segment_data.inity[i],2);
      segment_data.sgmxy += segment_data.initx[i] *
                            segment_data.inity[i];
       }
    mux  = segment_data.sgmx/4.0;
    muy  = segment_data.sgmy/4.0;
    muxx = segment_data.sgmx2 - Power(segment_data.sgmx,2)/4.0;
    muyy = segment_data.sgmy2 - Power(segment_data.sgmy,2)/4.0;
```

```
    muxy = segment_data.sgmxy - (segment_data.sgmx
                  * segment_data.sgmy) /4.0;
    if (-2.0 * muxy != 0 || muyy - muxx != 0)
     theta = (atan2( -2.0 * muxy, (muyy - muxx))) / 2.0;
    r = mux * cos(theta) + muy * sin(theta);
    for (j = 0; j < 4; ++j)
      {
       sds += Power(segment_data.initx[j] - mux,2) * Power(cos(theta),2);
       sds += Power(segment_data.inity[j] - muy,2) * Power(sin(theta),2);
       sds += 2.0 * (segment_data.initx[j] - mux) *
             (segment_data.inity[j] - muy)* cos(theta) * sin(theta);
      }
    segment_data.sgm_delta_sq = sds;
    segment_data.theta = theta;
    segment_data.r = r;
}

/*************************************************************************
   FUNCTION:       add_to_line()

   AUTHOR:         Mike Campbell

   DATE:           4 March 1996

   PURPOSE:        This procedure checks to see if the current return fit the
                   current line segment based on range and thinness ratio.

   RETURNS:        None. Sets line_status == 5 if cannot add return to current
                   segment.
*************************************************************************/
void add_to_line(x,y)

double x,y;

{
   double num_points;
   double m_major, m_minor, d_major, d_minor, theta, r;
   double mux, muy, muxx, muyy, muxy, sds;
   int i;

   num_points = (double)segment_data.num_points;
   segment_data.sgmx  += x;
   segment_data.sgmy  += y;
   segment_data.sgmx2 += Power(x,2);
   segment_data.sgmy2 += Power(y,2);
   segment_data.sgmxy += x * y;
   mux  = segment_data.sgmx / num_points;
   muy  = segment_data.sgmy / num_points;
   muxx = segment_data.sgmx2 - Power(segment_data.sgmx,2) / num_points;
   muyy = segment_data.sgmy2 - Power(segment_data.sgmy,2) / num_points;
   muxy = segment_data.sgmxy -
        (segment_data.sgmx*segment_data.sgmy) / num_points;
   m_major = (muxx+muyy)/2.0 -sqrt((muyy-muxx)*(muyy-muxx)/4.0 +
                                                    Power(muxy,2));
```

```
   m_minor = (muxx+muyy)/2.0 +sqrt((muyy-muxx)*(muyy-muxx)/4.0 +
                                                          Power(muxy,2));
   d_major = 4.0 * sqrt(fabs(m_minor/num_points));
   d_minor = 4.0 * sqrt(fabs(m_major/num_points));
   if ((d_minor / d_major) < thinness_requirement)
     {
      if (-2.0 * muxy != 0 || muyy - muxx != 0)
        theta = (atan2( -2.0 * muxy, (muyy - muxx))) / 2.0;
      r = mux * cos(theta) + muy * sin(theta);
      sds += Power(x - mux,2) * Power(cos(theta),2);
      sds += Power(y - muy,2) * Power(sin(theta),2);
      sds += 2.0 * (x - mux) * (y - muy) * cos(theta) * sin (theta);
      segment_data.sgm_delta_sq += sds;
      segment_data.theta = theta;
      segment_data.r = r;
      segment_data.endx = x;
      segment_data.endy = y;
      segment_data.d_major = d_major;
      segment_data.d_minor = d_minor;
      if((normal2(SONAR_725_bearing) > 336 ||
                        normal2(SONAR_725_bearing) < 024) &&
                                              SONAR_725_range < 5.0)
       {
        sprintf(Sonar_String_back,"COLLISION_THREAT");
        write(Sonar_to_OOD_fd[1],Sonar_String_back,MAXBUFFERSIZE);
        printf("%s\n",Sonar_String_back);
       }
     }
   else segment_data.line_status = 5;
 }

/*************************************************************************
   FUNCTION:       LINE_SEG *end_segment()

   AUTHOR:         Mike Campbell

   DATE:           4 March 1996

   PURPOSE:        This procedure finishes off a line segment placing it into
                   the LINE_SEG data structure, including the calculation of
                   the orientation of the line segment.

   RETURNS:        Current LINE_SEG.
*************************************************************************/
struct LINE_SEG *end_segment()


{
   struct LINE_SEG *seg_ptr;
   double startx, starty, endx, endy, delta, theta, r, length, t = 0;
   double bearing_end,bearing_start;

   startx = segment_data.startx;
   starty = segment_data.starty;
```

```
  endx   = segment_data.endx;
  endy   = segment_data.endy;
  theta  = segment_data.theta;
  r      = segment_data.r;
  delta  = startx * cos(theta) + starty * sin(theta) - r;
  startx = startx - (delta * cos(theta));
  starty = starty - (delta * sin(theta));
  delta  = endx * cos(theta) + endy * sin(theta) - r;
  endx   = endx - (delta * cos(theta));
  endy   = endy - (delta * sin(theta));
  length = sqrt(Power(startx - endx,2) + Power(starty - endy,2));
  seg_ptr = (struct LINE_SEG *) malloc (RECORD_SIZE);
  seg_ptr->headx  = startx;
  seg_ptr->heady  = starty;
  seg_ptr->tailx  = endx;
  seg_ptr->taily  = endy;
  seg_ptr->alpha  = theta;
  seg_ptr->start_time = segment_data.start_time;
  seg_ptr->finish_time = t;
  if( (endx-startx != 0 || endy-starty != 0) &&(endy != 0 || endx != 0) &&
          (startx != 0 || starty != 0))
  {
    bearing_end = atan2(endy,endx);
    bearing_start = atan2(starty,startx);
    if(((abs(quadrant(bearing_end) - quadrant(bearing_start)) != 3) &&
                  (bearing_end - bearing_start >= 0.0 )) ||
       ((abs(quadrant(bearing_end) - quadrant(bearing_start)) == 3) &&
                  (bearing_end - bearing_start < 0.0 )))
      seg_ptr->orientation  = atan2((endy-starty),(endx-startx));
    else
      seg_ptr->orientation  = atan2((starty-endy),(startx-endx));

  }
  seg_ptr->r      = r;
  seg_ptr->length = length;
  seg_ptr->dmajor = segment_data.d_major;
  seg_ptr->dminor = segment_data.d_minor;
  seg_ptr->next   = NULL;
  if((normal2(SONAR_725_bearing) > 336 ||
                  normal2(SONAR_725_bearing) < 024) &&
                                          SONAR_725_range < 5.0)
    {
     sprintf(Sonar_String_back,"COLLISION_THREAT");
     write(Sonar_to_OOD_fd[1],Sonar_String_back,MAXBUFFERSIZE);
     printf("%s\n",Sonar_String_back);
    }
  fprintf(out2,"# LINE SEGMENT\n%4lf %4lf %4lf %4lf\n%4lf %4lf %4lf
                       %4lf\n\n", seg_ptr->start_time, seg_ptr->headx,
                                  seg_ptr->heady, seg_ptr->orientation,
                     seg_ptr->finish_time, seg_ptr->tailx, seg_ptr->taily,
                                                    seg_ptr->orientation);
  fflush (out2);
  return seg_ptr;
}
```

```
/*************************************************************************
   FUNCTION:       reset_accumulators()

   AUTHOR:         Mike Campbell
   DATE:           4 March 1996

   PURPOSE:        This procedure resets all of the cumulative segment data,
                   preparing for a new segment to begin.
   RETURNS:        None.
*************************************************************************/
void reset_accumulators()
{
   segment_data.num_points =0.0;
   segment_data.sgmx = 0.0;
   segment_data.sgmy = 0.0;
   segment_data.sgmx2 = 0.0;
   segment_data.sgmy2 = 0.0;
   segment_data.sgmxy = 0.0;
}
/*************************************************************************
   FUNCTION:       add_circle()

   AUTHOR:         Mike Campbell

   DATE:           4 March 1996

   PURPOSE:        This procedure produces the circle representation of the
                   classified objects.

   RETURNS:        None.
*************************************************************************/

void  add_circle()


{
  char          newfile[22],oldfile[22];
  FILE          *new_circ_ptr;
  double        xtemp,ytemp,rtemp,radius;
  char          command[MAXLINE];
  double        length,headx,heady,tailx,taily,area,del_x,del_y,num_circles;
  int           i;

  headx = world[poly_num].headx;
  heady = world[poly_num].heady;
  tailx = world[poly_num].tailx;
  taily = world[poly_num].taily;
  area = world[poly_num].area;
  length = sqrt(fabs(((headx - tailx)*(headx - tailx) +
              (heady - taily) * (heady - taily))));
  sprintf(newfile, "circle_world.input%1i", poly_num);
  sprintf(oldfile, "circle_world.input%1i", poly_num - 1);
  sprintf(command,"cp %s %s", oldfile, newfile);
  if (poly_num > 0) system(command);
```

```
  if ((new_circ_ptr = fopen(newfile,"a")) == ((FILE *) 0))
    {
      printf("Error opening new world \n");
    }
   else
    {
    switch (world[poly_num].classification)
      {
       case 1:
        radius = global_radius;
        num_circles = ceil(length/global_radius);
        del_x = (tailx - headx) / num_circles;
        del_y = (taily - heady) / num_circles;
        for (i = 0; i < num_circles; ++i)
         {fprintf(new_circ_ptr,"Circle %6.4f %6.4f %6.4f %6.4f\n",
                 (headx + i*del_x),(heady + i*del_y),z,radius);
         }
        break;
      case 2:
        if ((sqrt(fabs(area*2/M_PI))) > 0.5)
              radius = sqrt(fabs(area*2/M_PI));
       else
              radius = 0.5;
       fprintf(new_circ_ptr,"Circle %6.4f %6.4f %6.4f %6.4f\n",
          (world[poly_num].centroidx / (world[poly_num].seg_count * 2)),
               (world[poly_num].centroidy / (world[poly_num].seg_count * 2)),
                           z,radius);
       break;
      case 3:
       if ((sqrt(fabs(area*2/M_PI))) > 0.5)
              radius = sqrt(fabs(area*2/M_PI));
       else
              radius = 0.5;
       fprintf(new_circ_ptr,"Circle %6.4f %6.4f %6.4f %6.4f\n",
         (world[poly_num].centroidx / (world[poly_num].seg_count * 2)),
                (world[poly_num].centroidy / (world[poly_num].seg_count * 2)),
                             z,radius);
       break;
      }
    }
  fclose(new_circ_ptr);
  sprintf(Sonar_String_back,"REPLAN %s",newfile);
  write(Sonar_to_OOD_fd[1],Sonar_String_back,MAXBUFFERSIZE);
  printf("%s\n",Sonar_String_back);

}

/*************************************************************************
   FUNCTION:       check_if_new()

   AUTHOR:         Mike Campbell

   DATE:           4 March 1996
```

```
   PURPOSE:        This procedure determines if the current object just
                   finished is a new object or correlates to an already
                   existing object.

   RETURNS:        "1" if it is a new object or "0" if it is not new.
*****************************************************************/

int check_if_new(new_poly)

struct Polyhedron new_poly;

{
  int  i,ns,s;
  double m1,m2,b1,b2;

  for (i=0; i < poly_num ; i++)
   {
       ns = new_poly.seg_count * 2;
       s  = world[i].seg_count * 2;
       if ((fabs(world[i].alpha - new_poly.alpha)) < 0.2 &&
                         world[i].classification == 1
            &&  (fabs(new_poly.centroidx/ns - world[i].centroidx/s) < 1.0) &&
            (fabs(new_poly.centroidy/ns - world[i].centroidy/s) < 1.0)
               && new_poly.classification == 1)
         {
          return 0;
       }
       else if ((fabs(new_poly.centroidx/ns - world[i].centroidx/s) < 1.0) &&
                (fabs(new_poly.centroidy/ns - world[i].centroidy/s) < 1.0) &&
                 (fabs(new_poly.area - world[i].area) < 5.0))
         {
          return 0;
       }
   }

  return 1;
}

/*****************************************************************
   FUNCTION:       classify_poly()

   AUTHOR:         Mike Campbell

   DATE:           4 March 1996

   PURPOSE:        This procedure classifies the objects based on size length
                   and other characteristics.

   RETURNS:        None.
*****************************************************************/

void classify_poly(n)

int n;
```

```
{
  double length,headx,heady,tailx,taily,area,alpha,del_x,del_y,num_circles;
  int    i;

  headx = world[n].headx;
  heady = world[n].heady;
  tailx = world[n].tailx;
  taily = world[n].taily;
  area = fabs(world[n].area);
  alpha = world[n].alpha;

  length = sqrt((headx - tailx)*(headx - tailx) +
            (heady - taily) * (heady - taily));
  if ((length > 5.0) && (area/length/length < 0.1))
    world[n].classification = 1;
  else if ((area >= 10.0) && (area <= 100.0))
    world[n].classification = 2;
  else
    world[n].classification = 3;
  if (check_if_new(world[n]) &&
       (length > 0.5 || world[n].seg_count >= 2 ))
    {
      add_circle();
    }
  else
   {
     reset_accumulators();
     poly_num -= 1;
   }

}



/*************************************************************************
   FUNCTION:      end_poly()

   AUTHOR:        Mike Campbell

   DATE:          4 March 1996

   PURPOSE:       This procedure ends the polyheron and classifies it.

   RETURNS:       None.
*************************************************************************/

void end_poly(ptr,n)

int n;
struct LINE_SEG *ptr;

{
  world[n].status = 1;
```

```
  classify_poly(n);
}


/*************************************************************************
   FUNCTION:       init_next_poly()

   AUTHOR:         Mike Campbell

   DATE:           4 March 1996

   PURPOSE:        This procedure initializes the next polyhedron number once
                   the current polyhedron is completed.

   RETURNS:        None.
*************************************************************************/
void init_next_poly(ptr,n)

int n;
struct LINE_SEG *ptr;

{
    double del_x,del_y;

    world[n].head_of_poly = (struct LINE_SEG *) malloc (RECORD_SIZE);
    world[n].head_of_poly->next = ptr;
    ptr->prev = world[n].head_of_poly;
    ptr->next = NULL;
    world[n].centroidx = (ptr->headx + ptr->tailx);
    world[n].centroidy = (ptr->heady + ptr->taily);
    world[n].seg_count = 1;
    world[n].area = 0.0;
    world[n].alpha = ptr->alpha;
    world[n].tailx = ptr->tailx;
    world[n].taily = ptr->taily;
    world[n].headx = ptr->headx;
    world[n].heady = ptr->heady;
    world[n].status = 0;
    del_x = world[n].headx - world[n].tailx;
    del_y = world[n].heady - world[n].taily;
    if (del_x*del_x + del_y*del_y >= 16.0)
      world[n].classification = 1; /*long enough to be a wall*/
    else
      world[n].classification = 3;  /*this means unknown*/

}


/*************************************************************************
   FUNCTION:       build_poly()

   AUTHOR:         Mike Campbell
```

```
   DATE:          4 March 1996

   PURPOSE:       This procedure builds polyhedron out of line segments.

   RETURNS:       None.
*************************************************************************/
void build_poly(ptr,n)

int n;
struct LINE_SEG *ptr;

{
   struct LINE_SEG *temp1 = NULL;
   double tri1,tri2,alpha1,alpha2,bearing1,bearing2;

  if (world[n].head_of_poly == NULL)
    {
      init_next_poly(ptr,n);
    }
  else
    {
     temp1 = world[n].head_of_poly->next;
     while (temp1->next != NULL)
       temp1 = temp1->next;
     if (((ptr->heady - temp1->taily) * (ptr->heady - temp1->taily) +
         (ptr->headx - temp1->tailx) * (ptr->headx - temp1->tailx)) > 36.0)
      {
       if (n == poly_num)
        {
         end_poly(ptr,n);
         poly_num += 1;
         init_next_poly(ptr,poly_num);
        }
     }
    else
       {
        while (temp1->next != NULL)
        temp1 = temp1->next; /* get to end of list */
          if (ptr->r > 0.0)
            alpha2 = ptr->alpha;
          else if (temp1->r > 0.0)
            alpha2 = normal(M_PI + ptr->alpha);
          else
            alpha2 = ptr->alpha;
          if (temp1->r > 0.0)
            alpha1 = temp1->alpha;
          else if (ptr->r > 0.0)
            alpha1 = normal(M_PI + temp1->alpha);
          else
            alpha1 = temp1->alpha;
          bearing1 = atan2(temp1->taily - y, temp1->tailx - x);
          bearing2 = atan2(ptr->taily - y, ptr->tailx - x);
        if ((fabs(normal(alpha2 - alpha1))) < 0.1745)
          {
```

```
      ptr->prev = temp1;
      temp1->next = ptr;
      ptr->next = NULL;
    world[n].centroidx += (ptr->headx + ptr->tailx);
    world[n].centroidy += (ptr->heady + ptr->taily);
    world[n].seg_count += 1;
    if (world[n].classification == 1)
      world[n].alpha = ((world[n].seg_count - 1)*world[n].alpha +
                        ptr->alpha)/world[n].seg_count;
    tri1 = triangle_area(world[n].headx,world[n].heady,
                world[n].tailx,world[n].taily,
                ptr->headx,ptr->heady);
    tri2 = triangle_area(world[n].headx,world[n].heady,
                ptr->headx,ptr->heady,ptr->tailx,ptr->taily);
    world[n].area += (tri1 + tri2);
    world[n].tailx = ptr->tailx;
    world[n].taily = ptr->taily;

  }
else if((abs(quadrant(bearing1) - quadrant(bearing2)) != 3) &&
           ((bearing1 - bearing2 > 0.0 )
              && (normaliz(alpha2 - alpha1) > 0.0)
           || ((bearing1 - bearing2 < 0.0)
              && (normaliz(alpha1 - alpha2) > 0.0))))
{
  ptr->prev = temp1;
  temp1->next = ptr;
  ptr->next = NULL;
  world[n].centroidx += (ptr->headx + ptr->tailx);
  world[n].centroidy += (ptr->heady + ptr->taily);
  world[n].seg_count += 1;
  tri1 = triangle_area(world[n].headx,world[n].heady,
                world[n].tailx,world[n].taily,
                ptr->headx,ptr->heady);
  tri2 = triangle_area(world[n].headx,world[n].heady,
                ptr->headx,ptr->heady,ptr->tailx,ptr->taily);
  world[n].area += (tri1 + tri2);
  world[n].tailx = ptr->tailx;
  world[n].taily = ptr->taily;
}
 else if((abs(quadrant(bearing1) - quadrant(bearing2)) == 3) &&
           ((bearing2 - bearing1 > 0.0 )
              && (normaliz(alpha2 - alpha1) > 0.0)
           || ((bearing2 - bearing1 < 0.0)
              && (normaliz(alpha1 - alpha2) > 0.0))))
  {
    ptr->prev = temp1;
    temp1->next = ptr;
    ptr->next = NULL;
    world[n].centroidx += (ptr->headx + ptr->tailx);
    world[n].centroidy += (ptr->heady + ptr->taily);
    world[n].seg_count += 1;
    tri1 = triangle_area(world[n].headx,world[n].heady,
                world[n].tailx,world[n].taily,
```

```
                                ptr->headx,ptr->heady);
            tri2 = triangle_area(world[n].headx,world[n].heady,
                                ptr->headx,ptr->heady,ptr->tailx,ptr->taily);
            world[n].area += (tri1 + tri2);
            world[n].tailx = ptr->tailx;
            world[n].taily = ptr->taily;
          }
        else
        {
          end_poly(ptr,n);
          poly_num += 1;
          init_next_poly(ptr,poly_num);
        }
      }
    }
}
```

```
/*************************************************************************
   FUNCTION:      print_list()

   AUTHOR:        Mike Campbell

   DATE:          4 March 1996

   PURPOSE:       This procedure outputs the final results for further
                  evalutation.

   RETURNS:       None.
*************************************************************************/
void print_list()

   {

    struct LINE_SEG *temp_ptr;

    for (i = 0; i < poly_num + 1; i++)
      {
      if (world[i].head_of_poly == NULL)
         temp_ptr = NULL;
      else
         temp_ptr = world[i].head_of_poly->next;
      while (temp_ptr != NULL)

      {
      fprintf(objectfile,"%4g %4g %4g\n%4g %4g %4g %4g\n",temp_ptr->headx,
```

```
        temp_ptr->heady,temp_ptr->orientation,
          temp_ptr->tailx,temp_ptr->taily,
          temp_ptr->alpha,temp_ptr->r);
      temp_ptr = temp_ptr->next;
      }
      fprintf(objectfile,"\n");
      /************ This section commented to facilitate plotting outputs

      fprintf(outfile,"Average centroidx is: %4g \n",
             (world[i].centroidx / (world[i].seg_count * 2)));
      fprintf(outfile,"Average centroidy is: %4g \n",
             (world[i].centroidy / (world[i].seg_count * 2)));
      fprintf(outfile,"Area is: %4g \n",world[i].area);
      fprintf(outfile,"Classification is: %i \n",world[i].classification);
      fprintf(outfile,"\n \n \n");*****************************************/
      fprintf(new_world,"%f %f %f %f %f %f %f %f %f %i %i %i\n",
      world[i].start,world[i].end,world[i].headx,
        world[i].heady,world[i].tailx,world[i].taily,
        world[i].centroidx,world[i].centroidy,world[i].area,
        world[i].seg_count,world[i].status,
        world[i].classification);
     }

   /*fprintf(outfile,"There are %2i objects \n",poly_num + 1);*/
}


/*This is sonar_globals.h */

  FILE      *worldfile,*new_world,*textfile,*outfile,*out2,*objectfile;
  char      line[132];
  int       i,i1,i2,i3,i4,i5,
            st_path,Rotate_count,Rotate_status,poly_num = 0,Sonar_mode=-1,
            one_bad_return,Search_status,bin_threshold,LOCATIONLAB = 0;
  double    t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12,
            x_search,y_search,SONAR_725_range,SONAR_725_bearing;
  double    step_size = 1.0, SONAR_725_strength, sigma_weighting = 3.0,
            residual_tolerance = 2.0, thinness_requirement =0.1, x_return,
            y_return;
  double    old_x,old_y,z,global_radius = 1;
  int       Scan_direction = -1; /* -1 is cw scan 1 is for ccw scan */

  typedef struct {
      double theta;
      double r;
      int num_points;
      int line_status;
      double initx[4];
      double inity[4];
      double sgm_delta_sq;
      double start_time;
      double startx;
      double starty;
      double endx;
```

```
      double endy;
      double sgmx;
      double sgmy;
      double sgmx2;
      double sgmy2;
      double sgmxy;
      double d_minor;
      double d_major;
   } SEG_DAT;

struct LINE_SEG
    {
     double      start_time,
                 finish_time;
     double      headx,
                 heady;
     double      tailx,
                 taily;
     double      alpha,
                 orientation,
                 r;
     double      length;
     double      dmajor,
                 dminor;
     struct      LINE_SEG *next;
     struct      LINE_SEG *prev;
    } ;
struct Polyhedron
    {
     double      start_time;
     double      end_time;
     struct      LINE_SEG  *head_of_poly;
     double      tailx,
                 taily;
     double      headx,
                 heady;
     double      centroidx,
                 centroidy;
     double      area;
     double      alpha;/*for walls only*/
     int         seg_count;
     int         status; /* 0 building 1 complete */
     int         classification; /*1=wall, 2=mine, 3=unknown*/
   };

struct      LINE_SEG *head = NULL;
SEG_DAT     segment_data;
struct      INE_SEG *finished_segment;
struct      Polyhedron world[100];
```

APPENDIX B. SOURCE CODE FOR SONAR COMMUNICATIONS

```
/*************************************************************************
   FILENAME:       sonar_comms.c

   AUTHOR:         Mike Campbell
                   Sonar Communication Code:  Modified from Dave Marco's Code
                   Serial Port Initialization Code:Modified from Dave
                                                          McClarin's Code
   DATE:           22 January 1996

   PURPOSE:        Handle all communications with the sonar system including:
                   initialization and pinging sonar.

   REVISION:       This code is constantly being improved and expanded current
                   revision is available at:
                   http://www.stl.nps.navy.mil/~auv/tactical/

   FUNCTIONS:      int initialize_sonar_serial();
                   int initialize_sonar_systems();
                   double  Ping_Sonar();
                   void initialize_sonar();
                   char set_scanning_gain();
                   void center_sonar();
                   char send_command();
                   int read_port();
*************************************************************************/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <fcntl.h>
#include <ctype.h>
#include <errno.h>
#include <string.h>
#include <stropts.h>
#include <sys/conf.h>
#include <sys/stat.h>
#include <unistd.h>
#include "sonar_globals.h"
#include "termiox.h"
#include <sys/uio.h>
#include <termio.h>

FILE *outfp;
double sonar_heading;
int initialize_sonar_serial();
int initialize_sonar_systems();
double  Ping_Sonar();
void initialize_sonar();
char set_scanning_gain();
void center_sonar();
char send_command();
```

```
int read_port();
/*************************************************************************
   FUNCTION:      initialize_sonar_systems()

   AUTHOR:        Mike Campbell

   PURPOSE:       Initializes the system including the serial port and
                  sonar head.

   RETURNS:       One if it is successful, zero if not
*************************************************************************/

int initialize_sonar_systems()

{
   char c[1],x1[50],x2[50],x3[50],x[200],y[1];
   char *t;
   unsigned short bin_byte,dummy_byte,bin0,bin1,bin2,bin3;
   char buffer[100];
   int i,j,k,n,w_path,r_path,n_loops,n_bytes,nnn;
   int RESET_PORT,timeout;
   int n_b;

   char st725_mode;                /* 'S' for Scanning, 'P' for Profiling */
   int st725_Nbins;                /* No. of bins to collect 64 or 128 */
   int st725_max_range;
   int st725_power;
   int st725_Ecpuls;                /* N/A */
   int st725_psi_sonar_count;
   double st725_psi_sonar;
   int st725_sweep_sign;
   int st725_ssiz;
   if ((outfp = fopen ("raw_data.sonar", "w")) ==((FILE *) 0))
   {
      printf("Error opening out file \n");
      exit(0);
   }
   st725_mode                 = 'S';
   st725_psi_sonar_count      = 0;
   st725_psi_sonar            = 0.0;
   st725_sweep_sign           = 1;
   st725_ssiz                 = 1;
   st725_max_range            = 6;
   st725_power                = 12;
   r_path                     = initialize_sonar_serial("/dev/ttya");

   initialize_sonar(r_path,st725_mode,st725_max_range,st725_power,
                    &st725_Nbins,st725_ssiz,st725_Ecpuls);
   center_sonar(r_path);
   sonar_heading = 0.0;
   return r_path;
}
```

```
/*************************************************************************
   FUNCTION:       Ping_Sonar_()

   AUTHOR:         Mike Campbell

   PURPOSE:        Pings the sonar and scans in the Scan_direction, reads
                   the sonar response and calculates Range ignoring first
                   bin_threshold bins

   RETURNS:        Returns range in feet and sets SONAR_725_strength

*************************************************************************/

double  Ping_Sonar(r_path,Scan_direction,bin_threshold)

int r_path,Scan_direction,bin_threshold;

{
  char c[1],x[200],y[1];
  int Intensity[64],i,max_range = 0,max_loc = 0;
  char *t;
  unsigned short bin_byte,bin0,bin1,bin2,bin3;
  int k,n,num_bytes;
  double range;

  switch(Scan_direction)
   {
    case -1:
     c[0] = 'R';
     sonar_heading += 0.9;
     break;
    case 0:
     c[0] = 'S';
     break;
    case 1:
     c[0] = 'L';
     sonar_heading -= 0.9;
     break;
   }
  write(r_path,c,1); /* write characters to path /t2 dev. */
  ioctl(r_path,I_NREAD,&num_bytes);
  for(k=0;k<16;++k) /* Read First 8 Bytes */
   {
     t = &bin_byte;
     n=read(r_path,x,1);
     t[0] = x[0];
     n=read(r_path,x,1);
     t[1] = x[0];

     bin3 = bin_byte << 12;
     bin3 = bin3 >> 12;
```

```
    bin2 = bin_byte << 8;
    bin2 = bin2 >> 12;

    bin1 = bin_byte << 4;
    bin1 = bin1 >> 12;

    bin0 = bin_byte >> 12;

    Intensity[k*4] = bin0;
    Intensity[k*4 + 1] = bin1;
    Intensity[k*4 + 2] = bin2;
    Intensity[k*4 + 3] = bin3;
    if (k > 0)
  }

   n=read(r_path,x,1);
   for(i = bin_threshold; i < 64; i++)
       { if(Intensity[i] > max_range + 2 && Intensity[i] > 7)
         {
          max_range = Intensity[i];
          max_loc = i;
         }
        }
   if(max_loc > 0)
    {
     range = 0.5126*(1 + max_loc); /*conversion from bin # to range in feet
                                          set for 10 meter scale*/
     SONAR_725_strength = max_range;
    }
   else
    {
     range = 0;
     SONAR_725_strength = 0;
    }
  return range;
}
/*************************************************************************
   FUNCTION:       initialize_sonar_serial()

   AUTHOR:         Mike Campbell  Modified from Code of Dave McClarin

   PURPOSE:        Initializes serial port /dev/ttya

   RETURNS:        0 if failed and 1 if succeeds
*************************************************************************/
int initialize_sonar_serial(char port[50])
{
   int i, path, stat;

   struct termio term;

   if ((path = open(port, O_RDWR|O_NONBLOCK)) == -1)
    {
      fprintf(stderr,"No serial connection.\n");
```

```
    }
   else
    {
      path = open(port, O_RDWR);
      memset(&term, 0, sizeof(term));
      term.c_iflag = IXOFF;
      term.c_oflag = 0;
      term.c_cflag = B9600|CS8|CLOCAL|CREAD|HUPCL;
      term.c_lflag = 0;
      term.c_line = 0;
      term.c_cc[VMIN] = 0;
      term.c_cc[VTIME] = 1;
      if (ioctl(path, TCSETAF, &term) == -1)
         fprintf(stderr,"Unable to set port parameters.\n");
   }
   return path;
}
/*************************************************************************
   FUNCTION:      read_port()

   AUTHOR:        Mike Campbell  Modified from Code of Dave Marco

   PURPOSE:       Reads the serial port

   RETURNS:       Number of bytes read

*************************************************************************/

int read_port(int port_fd, int num_bytes, char *data)
{
   const int MAX_RETRIES = 100;
   int count =0;
   int num_tries = 0;
   int nbr;

   while ((count < num_bytes) &&
          (num_tries < MAX_RETRIES) &&
          ((nbr = read(port_fd, &data[count], 1)) != -1)) {
      num_tries++;
      count += nbr;
   }

   if (count != num_bytes) {
         if (num_tries == MAX_RETRIES)
            fprintf(stderr, "Error: too many retries reading serial port\n");
         else
            fprintf(stderr, "Error: serial port read failed\n");
         fprintf(stderr,"  in read_port\n");
   }

   return count;
}

/*************************************************************************
```

```
   FUNCTION:       send_command()

   AUTHOR:         Mike Campbell  Modified from Code of Dave Marco

   PURPOSE:        This function sends a single character and reads back a
single
                   character from the sonar
   RETURNS:        char read from sonar

*****************************************************************************/

char send_command(path,command)
   int path;
   char command;
{

   unsigned n,n_bytes=0;
   char reply,x[20],c[1];
   c[0] = command;

   n = write(path,c,1);
   while(n_bytes != 1)
   ioctl(path,I_NREAD,&n_bytes);
   if(n_bytes == 1)
   {
      n = read(path,x,n_bytes);
   }
   reply = x[0];
   return(reply);
}

/*****************************************************************************
   FUNCTION:       initialize_sonar()

   AUTHOR:         Mike Campbell  Modified from Code of Dave Marco

   PURPOSE:        Initializes sonar head to desired parameters

   RETURNS:        None
*****************************************************************************/

void initialize_sonar(path,mode,max_range,gain,nbins,ssiz,Ecpuls)

   int path;            /* Path opened for particular head */
   char mode;           /* 'S' = Scanning mode, 'P' = Profiling mode */
   int max_range;       /* 1, 2, 4, 6, 10, 20, 25, 30, 50, or 100 meters */
   int gain;            /* 0 <= gain <= 100 */
   int *nbins;          /* # of bins to collect 64 or 128 */
   int ssiz;            /* 1, 2, or 4 = 0.9, 1.8, or 3.6 deg. */
   int Ecpuls;
{

   unsigned n_bytes;
   unsigned short T1,T2,T3,Tchecksum;
```

```
   char c[1],x[20],*t,y[1],reply;
   unsigned short byte,byte1,Nsampl,Nbins,Range_Code,checksum;
   unsigned short TxPulseMSByte,TxPulseLSByte,Gecmin,Rng_unt;
   int i,j,k,n,word,TxPulse;

   int Timout,Lokout,Eswait,Gaindt,Ecsclx,Ecscly;
   int Maxdst,Dacscx,Dacscy;
   int EchoSounder[10];
/*************************************************************************
  INITIALIZATIN PARAMETERS FOR SCANNING MODE (ST-725 AND ST-1000 HEADS)

  Range  | TxPulse     | NSAMPL   |  NBINS   | Range Code | checksum
  meters | 1.96 usec   |          |          |            | Lower 8 bits of
sum
         |  dec    hex | dec  hex | dec  hex | dec   hex  | dec  hex
    6    |  30    001E |   1   01 |  64   40 |   0    00  |  95   5F
   10    |  30    001E |   3   03 |  64   40 |   1    01  |  98   62
   20    |  100   0064 |   3   03 | 128   80 |   2    02  | 233   E9
   25    |  125   007D |   4   04 | 128   80 |   3    03  |   4   04
   30    |  150   0096 |   6   06 | 128   80 |   4    04  |  32   20
   50    |  250   00FA |  12   0C | 128   80 |   5    05  | 139   8B
  100    |  475   01DB |  26   1A | 128   80 |   7    07  | 125   7D
*************************************************************************
  INITIALIZATIN PARAMETERS FOR PROFILING MODE (ST-1000 HEAD)

  Range  | TxPulse | NSAMPL | NBINS | Range Code | TIMOUT | Maxdst
  meters |         |        |       |            |        |
         |         |        |       |            |        |
    1    |   30    |   1    |   64  |    00      |  1500  | 1500
    2    |   30    |   1    |   64  |    01      |  3000  | 3000
    4    |   30    |   1    |  128  |    02      |  6000  | 6000
    6    |   30    |   1    |  128  |    03      |  9000  | 9000
   10    |   40    |   1    |  128  |    04      | 15000  | 15000
   20    |   50    |   3    |  128  |    05      | 30000  | 30000
   30    |   75    |   6    |  128  |    06      | 45000  | 45000
   50    |  100    |  12    |  128  |    07      | 65535  | 65535

   ECPULS  = 30
   LOKOUT  = 200
   ESWAIT  = 25600
   GECMIN  = Byte
   GAINDT  = 64
   ECSCLX  = 16383
   ECSCLY  = 11374
   DACSCX  = 256
   DACSCY  = 3125
   Rng Unt = 1
*************************************************************************/
   if(mode == 'S') /* Set up for Scanning */
   {
      switch(max_range)
      {
         case 6:
            TxPulse    = 30;
```

```
   Nsampl      =  1;
   Nbins       = 64;
   Range_Code  =  0;
   T1          =  3;
   T2          = 65;
   T3          = 12;
   Tchecksum   = 80;
break;

case 10:
   TxPulse     = 30;
   Nsampl      =  3;
   Nbins       = 64;
   Range_Code  =  1;
   T1          =  3;
   T2          = 65;
   T3          = 20;
   Tchecksum   = 88;
break;

case 20:
   TxPulse     = 100;
   Nsampl      =   3;
   Nbins       = 128;
   Range_Code  =   2;
   T1          =   3;
   T2          = 129;
   T3          = 40;
   Tchecksum   = 172;
break;

case 25:
   TxPulse     = 125;
   Nsampl      =   4;
   Nbins       = 128;
   Range_Code  =   3;
   T1          =   3;
   T2          = 129;
   T3          =  50;
   Tchecksum   = 182;
break;

case 30:
   TxPulse     = 150;
   Nsampl      =   6;
   Nbins       = 128;
   Range_Code  =   4;
   T1          =   3;
   T2          = 129;
   T3          =  60;
   Tchecksum   = 192;
break;

case 50:
```

```
         TxPulse    = 250;
         Nsampl     =  12;
         Nbins      = 128;
         Range_Code =   5;
         T1         =   3;
         T2         = 129;
         T3         = 100;
         Tchecksum  = 232;
      break;

      case 100:
         TxPulse    = 475;
         Nsampl     =  26;
         Nbins      = 128;
         Range_Code =   7;
         T1         =   3;
         T2         = 129;
         T3         = 200;
         Tchecksum  = 76;
      break;
   } /* End switch */
}
else if(mode == 'P') /* Set up for Profiling */
{
   switch(max_range)
   {
      case 1:
         TxPulse    = 30;
         Nsampl     =  1;
         Nbins      = 64;
         Range_Code =  0;

         Timout     = 1500;
         Maxdst     = 1500;
      break;

      case 2:
         TxPulse    = 30;
         Nsampl     =  1;
         Nbins      = 64;
         Range_Code =  1;

         Timout     = 3000;
         Maxdst     = 3000;
      break;

      case 4:
         TxPulse    =  30;
         Nsampl     =   1;
         Nbins      = 128;
         Range_Code =   2;

         Timout     = 6000;
         Maxdst     = 6000;
```

```
      break;

      case 6:
         TxPulse    =  30;
         Nsampl     =   1;
         Nbins      = 128;
         Range_Code =   3;

         Timout     = 9000;
         Maxdst     = 9000;
      break;

      case 10:
         TxPulse    =  40;
         Nsampl     =   1;
         Nbins      = 128;
         Range_Code =   4;

         Timout     = 15000;
         Maxdst     = 15000;
      break;

      case 20:
         TxPulse    =  50;
         Nsampl     =   3;
         Nbins      = 128;
         Range_Code =   5;

         Timout     = 30000;
         Maxdst     = 30000;
      break;

      case 30:
         TxPulse    =  75;
         Nsampl     =   6;
         Nbins      = 128;
         Range_Code =   6;

         Timout     = 45000;
         Maxdst     = 45000;
      break;

      case 50:
         TxPulse    = 100;
         Nsampl     =  12;
         Nbins      = 128;
         Range_Code =   7;

         Timout     = 65535;
         Maxdst     = 65535;
      break;
   } /* End switch */

   /* Values Common to all Ranges */
```

```
   /*Ecpuls     = 75;*/
   /*printf("Input Ecpuls\n");
   scanf("%d",&Ecpuls);
   printf(" \n");
   printf("******* Ecpuls = %d\n",Ecpuls);
   printf(" \n");*/
   Lokout     = 200;
   Eswait     = 25600;
   Gecmin     = 2.55*gain;
   /*printf(" \n");
   printf("******* Gecmin = %d\n",Gecmin);
   printf(" \n");*/
   Gaindt     = 64;
   Ecsclx     = 16383;
   Ecscly     = 11374;
   Dacscx     = 256;
   Dacscy     = 3125;
   Rng_unt    = 1;

}
else
{
   printf("Wrong mode!\n");
}
*nbins = Nbins;

/* Send Sonar Parameters */
c[0] = 'P';
write(path,c,1);

/* Send TxPulse Length (Word) in 1.96 usec units */
word = TxPulse & 0x00ff; /* Byte = LSByte of TxPulse */
byte = word;
TxPulseLSByte = byte;

y[0] = (char) byte;
n = write(path,y,1);    /* Send LSByte First */

word = TxPulse >> 8;    /* Byte = MSByte of TxPulse */
byte = word;
TxPulseMSByte = byte;
y[0] = (char) byte;
n = write(path,y,1);    /* Send MSByte Last */

/* Send NSAMPL (Byte) N0. A/D Samples per Bin */
byte = Nsampl;
y[0] = (char) byte;
n = write(path,y,1);

/* Send NBINS (Byte) No. of Bins to Collect */
byte = Nbins;
y[0] = (char) byte;
n = write(path,y,1);
```

```
 /* Send Range Code 0-8 (obsolete) (Byte) */
 byte = Range_Code;
 y[0] = (char) byte;
 n = write(path,y,1);

 /* Send DataByte checksum (Byte). Should be the lowest 8 bits of */
 /*   the sum of all Bytes */
 word = TxPulseMSByte + TxPulseLSByte + Nsampl + Nbins + Range_Code;
 word = word & 0x00ff; /* Mask MSByte to get last 8 bits for checksum; */
 checksum = word;
byte = checksum;
 y[0] = (char) byte;
 n = write(path,y,1);

 sleep(1);
 ioctl(path,I_NREAD,&n_bytes);
 /* Read Reply to Checksum */
 n = read(path,x,1);
 reply = x[0];
 if(reply == 'T')
 {
    printf("Parameter Checksum Ok\n");
 }
 else
 {
    printf("Parameter Checksum INCORRECT!!!\n");
 }

 /* Enable halfstep should reply 'H' */
 reply = send_command(path,'H');
 if(reply == 'H')
 {
    printf("Half step set\n");
 }
 else
 {
    printf("Half step not set!\n");
 }

 /* Enable TVG should reply 'X' */
 reply = send_command(path,'X');
 if(reply == 'X')
 {
    printf("TVG set\n");
 }
 else
 {
    printf("TVG not set!\n");
 }

 /* Set mode return Range bin Peak should reply 'K' */
 reply = send_command(path,'K');
 if(reply == 'K')
```

```
{
   printf("Range bin Peak mode Ok\n");
}
else
{
   printf("Range bin Peak mode not set!\n");
}

/* Set Final Gain for TVG, should reply 'E' */
reply = set_scanning_gain(path,83,'E');
if(reply == 'E')
{
   printf("Final TVG Gain set\n");
}
else
{
   printf("Final TVG Gain not set!\n");
}

if(mode == 'S') /* Scanning mode */
{
   /* Set Initial Gain for TVG, should reply 'C' */
   reply = set_scanning_gain(path,gain,'C');
   if(reply == 'C')
   {
      printf("Initial TVG Gain set\n");
   }
   else
   {
      printf("Initial TVG Gain not set!\n");
   }
}
else /* Profiling mode */
{

   EchoSounder[0] = Ecpuls;
   EchoSounder[1] = Timout;
   EchoSounder[2] = Lokout;
   EchoSounder[3] = Eswait;
   EchoSounder[4] = Gaindt;
   EchoSounder[5] = Ecsclx;
   EchoSounder[6] = Ecscly;
   EchoSounder[7] = Maxdst;
   EchoSounder[8] = Dacscx;
   EchoSounder[9] = Dacscy;

   checksum = 0;

   /* Send Profiler Sonar Parameters */

   c[0] = 'J';
   write(path,c,1);

   /* Send First 4 Parameters (Words) */
```

```
for(i=0;i<4;++i)
{
   word = EchoSounder[i] & 0x00ff; /* Byte = LSByte of EchoSounder[i] */
   byte = word;
   checksum = checksum + byte; /* Add up the checksum */
   y[0] = (char) byte;
   n = write(path,y,1);     /* Send LSByte First */

   word = EchoSounder[i] >> 8;     /* Byte = MSByte of EchoSounder[i] */
   byte = word;
   checksum = checksum + byte; /* Add up the checksum */
   y[0] = (char) byte;
   n = write(path,y,1);     /* Send MSByte Last */
}

/* Send Gecmin (Byte) */
byte = Gecmin;
checksum = checksum + byte;
y[0] = (char) byte;
n = write(path,y,1);

/* Send Last 6 Parameters (Words) */
for(i=4;i<10;++i)
{
   word = EchoSounder[i] & 0x00ff; /* Byte = LSByte of EchoSounder[i] */
   byte = word;
   checksum = checksum + byte; /* Add up the checksum */
   y[0] = (char) byte;
   n = write(path,y,1);     /* Send LSByte First */

   word = EchoSounder[i] >> 8;     /* Byte = MSByte of EchoSounder[i] */
   byte = word;
   checksum = checksum + byte; /* Add up the checksum */
   y[0] = (char) byte;
   n = write(path,y,1);     /* Send MSByte Last */
}
/* Send Rng_unt (Byte) */
byte = Rng_unt;
checksum = checksum + byte;
y[0] = (char) byte;
n = write(path,y,1);

/* Send DataByte checksum (Byte). Should be the lowest 8 bits of  */
/*   the sum of all Bytes */
checksum = checksum & 0x00ff; /* Mask MSByte to get last 8 bits */

printf("Profile checksum = %d\n",checksum);
byte = checksum;
y[0] = (char) byte;
n = write(path,y,1);

sleep(1);
ioctl(path,I_NREAD,&n_bytes);
/* Read Reply to Checksum */
```

```
      n = read(path,x,1);
      reply = x[0];
      if(reply == 'T')
      {
         printf("Profile Checksum Ok\n");
      }
      else
      {
         printf("Profile Checksum Incorrect\n");
      }
   }
   /* Check if head is using default settings. Reply is 'T' if yes, */
   /* 'F' if not */
   reply = send_command(path,'D');
   if(reply == 'T')
   {
      printf("Head is still using Default Settings!\n");
   }
   else
   {
      printf("Head not using Default Settings\n");
   }
 }

/*************************************************************************
   FUNCTION:      center_sonar()

   AUTHOR:        Dave Marco

   PURPOSE:       Function to set sonar gain, 0 <= gain <= 100

   RETURNS:       none
*************************************************************************/
char set_scanning_gain(path,gain,which_gain)

int   path,gain;
char  which_gain; /* which_gain = 'B' for Initial, 'E' for Final */
{
   unsigned short byte;
   unsigned n,n_bytes;
   char reply,y[1],x[20],c[1];

   /* Set Initial or Final Gain for TVG should reply */
   /* 'C' for Initial or 'E' for Final */
   c[0] = which_gain;
   write(path,c,1);
   byte = 2.55*gain;
   y[0] = (char) byte;
   n = write(path,y,1);
   sleep(1);
   ioctl(path,I_NREAD,&n_bytes);
   read(path,x,n_bytes);
   reply = x[0];
   return(reply);
```

```
}

/*************************************************************************
   FUNCTION:       center_sonar()

   AUTHOR:         Dave Marco

   PURPOSE:        centers the sonar head

   RETURNS:        none
*************************************************************************/
void center_sonar(path)

   int path;
{
   int i;
   int direction,encoder_width;
   char encode;

   encode='A';
   encoder_width = 0;
   direction = 1;
   printf("Inside center\n");
   /* Clear out any junk from buffer at startup */
   while(encode != '3')
   {
      printf("encode = %c path = %d\n",encode,path);
      encode = send_command(path,'V');
   }

     /* Are we inside the Encoder Sensor ? */

   encode = send_command(path,'M'); /* Test Head Direction (No Step) */

   if((encode == 't') || (encode == 'T'))
   {
      while( (encode == 't') || (encode == 'T') )
      {
       encode = send_command(path,'+'); /* Index Sonar '+' direction */
      }
      /* Outside Encoder Sensor Now */
      direction = -1; /* Reverse Sonar Rotation to Establish Encoder Width */
   }

   while( (encode == 'f') || (encode == 'F') )
   {
      if(direction == 1)
      {
       encode = send_command(path,'+'); /* Index Sonar '+' direction */
      }
      else
      {
       encode = send_command(path,'-'); /* Index Sonar '-' direction */
      }
```

```
   }
   /* Found Edge of Encoder */
   while( (encode == 't') || (encode == 'T') )
   {
      encoder_width = encoder_width + 1;
     if(direction == 1)
      {
       encode = send_command(path,'+'); /* Index Sonar '+' direction */
      }
      else
      {
       encode = send_command(path,'-'); /* Index Sonar '-' direction */
      }
   }

/* If direction = +1, Go Back 5 Steps to Establish Center
     If direction = -1, Go Forward 5 Steps to Establish Center */

   if(direction == 1)
   {
      for(i=0;i<5;++i)
      {
        encode = send_command(path,'-');
      }
   }
   if(direction == -1)
   {
        for(i=0;i<5;++i)
        {
              encode = send_command(path,'+');
        }
   }
   printf("Center Established\n");
   sonar_heading = 0.0;
}
```

APPENDIX C. CODE FOR SONAR GNUPLOTS

```
##############################################################################
#                                                                            #
#       filename:  excerpt from auv_plot_1_second.gnu                        #
#                                                                            #
#       function:  GNUPLOT V3.5 script to plot AUV telemetry data            #
#                  to screen & to PostScript files                           #
#                                                                            #
#        updated:  12 March 96                                               #
#                                                                            #
#         author:  Don Brutzman, excerpt written by Mike Campbell            #
#                                                                            #
#      execution:  gnuplot> load "auv_plot_1_second.gnu"                     #
#                  gnuplot> reread                                           #
#                                                                            #
#                  unix> gnuplot  auv_plot_1_second.gnu                      #
#                                                                            #
#    re-plotting:                                                            #
#       'xpsview'  -wp -skipc -or landscape ~/execution/AUV_telemetry.ps &   #
#       ghostview                -landscape ~/execution/AUV_telemetry.ps &   #
#                                                                            #
# C program call:  system ("gnuplot auv_plot_1_second.gnu");                 #
#                                                                            #
#      telemetry:  mission.output.telemetry (AUV telemetry 0.1 sec interval)#
#      alternate:  mission.output.1_second  (AUV telemetry 1.0 sec interval)#
#                                                                            #
#  original plot:  unix> gnuplot  auv_plot.gnu                               #
#                                                                            #
#   output files:  AUV_telemetry.ps & *.eps plots                            #
#                                                                            #
#  related files:  execution.c                                               #
#                  underwater virtual world                                  #
#                                                                            #
# output archive:  ftp://taurus.cs.nps.navy.mil/pub/auv/AUV_telemetry.ps.Z   #
#                                                                            #
#    gnuplot FAQ:    ftp://ftp.dartmouth.edu/pub/gnuplot/faq/gpt_faq.html    #
                                                                          #
#   distribution:    http://www.cs.dartmouth.edu/gnuplot/                    #
#                                                                            #
##############################################################################


# setup:

set terminal x11  # gnuplot version 3.4  (no auto redraw)

# set terminal iris4d   # gnuplot version 3.5  (x11 is OK)

set time
set grid
set data style linespoints

set samples 10         # data point plotting frequency
```

```
##############################################################################

set xlabel "East -> (y_world) [ft]"
set ylabel "North ^ (x_world) [ft]"


pause -1 "hit enter to continue with sonar plots"

# add sonar here

set title  "NPS AUV sonar outputs" 26,.8
plot "../tactical/data_points.sonar" title "processed sonar returns" with
points
pause -1 "hit enter to continue with sonar plot 2 "

set title  "NPS AUV sonar outputs" 26,.8
plot "../tactical/line_segments.sonar" using 2:3  title "fitted line segments"
pause -1 "hit enter to continue with sonar plot 3 "

set title  "NPS AUV sonar outputs" 26,.8
plot "../tactical/data_points.sonar" with points, \
     "../tactical/line_segments.sonar"  using 2:3 title "line segments over
data points"
pause -1 "hit enter to continue with sonar plot 4 "

set title  "NPS AUV sonar outputs" 26,.8
plot "../tactical/line_segments.sonar" using 2:3, "../tactical/objects.sonar"
title "objects imposed over line segments"
pause -1 "hit enter to continue with sonar plot 5 "

set title  "NPS AUV sonar outputs" 26,.8
plot "../tactical/objects.sonar" title "objects built from sonar data"
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
   8725 John J. Kingman Rd., STE 0944
   Alexandria, Virginia 22060-6218

2. Dudley Knox Library.....2
   Naval Postgraduate School
   Monterey, California 93943-5101

3. Computer Technology Programs, Code CS.....1
   Naval Postgraduate School
   Monterey, California 93943-5118

4. Chairman, Code EC 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California 93943-5121

5. Dr. Donald P. Brutzman, Code UW/Br.....2
   Undersea Warfare Academic Group
   Naval Postgraduate School
   Monterey, California 93943-5126

6. Dr. Xiaoping Yun, Code EC/ YX.....2
   Department Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California 93943-5121

7. Dr. Anthony J. Healey, Code ME/Hy.....1
   Mechanical Engineering Department
   Naval Postgraduate School
   Monterey, California 93943-5146

8. Dr. Robert McGhee, Code CS/Mz.....1
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943-5118

9. CDR Michael J. Holden, USN, Code CS/Hm.....1
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943-5118

10. Dr. Yutaka Kanayama, Code CS/Ka 1
Computer Science Department
Naval Postgraduate School
Monterey, California 93943-5118

11. Dr. Ted Lewis, Code CS 1
Chair, Computer Science Department
Naval Postgraduate School
Monterey, California 93943-5118

12. Russ Whalen, Code CS 1
Computer Science Department
Naval Postgraduate School
Monterey, California 93943-5118

13. Dave Marco, Code ME/Ma 1
Mechanical Engineering Department
Naval Postgraduate School
Monterey, California 93943-5146

14. Dr. Lawrence Ziomek, Code EC/Zm 1
Department Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California 93943-5121

15. Commander, Naval Undersea Warfare Center Division 1
1176 Howell Street
Attn: Erik Chaum, Code 2251, Building 1171-3
Combat Systems Engineering and Analysis Laboratory (CSEAL)
Newport, Rhode Island 02841-1708

16. Dr. James Bellingham 1
Underwater Vehicles Laboratory, MIT Sea Grant College Program
292 Main Street
Massachusetts Institute of Technology
Cambridge, Massachusetts 02142

17. Dr. John Leonard 1
Underwater Vehicles Laboratory, MIT Sea Grant College Program
292 Main Street
Massachusetts Institute of Technology
Cambridge, Massachusetts 02142

18. Kevin Gomes.....1
4425 Archwood Dr.
Colorado Springs, Colorado 80920

19. LT Michael S. Campbell.....1
291 Welcome Way
Carlisle, Ohio 45005